Editor version available on Springer website at:
https://link.springer.com/article/10.1007/s10270-017-0588-7
DOI: 10.1007/s10270-017-0588-7

A Framework to Specify System Requirements using Natural Interpretation of UML/MARTE Diagrams

Aamir M. Khan · Frédéric Mallet · Muhammad Rashid

Received: date / Accepted: date

Abstract The ever increasing design complexity of embedded systems is constantly pressing the demand for more abstract design levels and possible methods for automatic verification and synthesis. Transforming a text-based user requirements into semantically sound models is always difficult and error prone as mostly these requirements are vague and improperly documented. This paper presents a framework to specify textual requirement graphically in standard modeling artifacts like UML and MARTE in the form of temporal patterns. The underlying formal semantics of these graphical models allow to eliminate ambiguity in specifications and automatic design verification at different abstraction levels using these patterns. The semantics of these temporal operators are presented formally as state automatons and a comparison is made to the existing CCSL relational operators. To reap the benefits of MDE, a software plugin *TemPAC* is presented as part of the framework to transform the graphical patterns into CCSL and Verilog-based observers.

Keywords FSL · Graphical Properties · UML · MARTE · CCSL · Modeling · Embedded Systems

1 Introduction

Conventionally, the design of an embedded system starts with the system requirements specified by the requirements engineers. These requirements are usually in the form of natural language sentences mentioning the different design components, parameters, and constraints. The next step in Electronic Design Automation (EDA) domain is to build an executable model to implement the requirements and perform early validation. For instance, languages like SystemC [1] are often used to build models at the Electronic System Level (ESL) [2] in the early design phases. While the steps after ESL down to transaction level (TLM) or register transfer level (RTL) models have been well covered in the literature, there is a big gap between the early informal natural language requirements and ESL models. Intermediate levels like the Formal Specification Level (FSL) [3–5] have been proposed to fill this gap with models that are both close enough to requirements engineer concerns, and formal enough to allow further phases of automatic or semi-automatic generation and verification.

This paper contributes to this effort at FSL. Our solution attempts to reuse as much as possible the Unified Modeling Language (UML) [6] and some of its extensions. Indeed, the UML is a well-accepted modeling language which provides facilities to develop models at different abstraction levels. As a general purpose language, UML needs to be tailored when addressing specific domains. Selic et al. [7] recommend

Partially funded by the National Science, Technology, and Innovation Plan (NSTIP) Saudi Arabia.

A. M. Khan (✉)
University of Buraimi, Buraimi, Oman
E-mail: Aamir.m@uob.edu.om

F. Mallet
Université Nice Sophia Antipolis, INRIA Méditerranée, Sophia Antipolis, France
E-mail: Frederic.Mallet@inria.fr

a joint use of MARTE [8] and SysML [9] to build timed and untimed requirements for real-time and embedded systems. We follow this recommendation. Additionally, MARTE proposes as an annex, the Clock Constraint Specification Language (CCSL) [10] to complement UML/MARTE modeling elements with timed or causal extensions. CCSL is also used to encode the semantics of UML/MARTE models and resolve potential semantic variation points [11]. In our proposal, CCSL is important to keep the requirements formal and executable. We promote the use of UML-based models over the use of temporal logics formula, since they have a wider acceptance in industry. Indeed, while temporal logics, like LTL/CTL [12], are widely used in the latter stages in conjunction with model-checkers [13], they are not suitable to directly express high-level requirements at early stages and are commonly rejected by requirement engineers [14, 15] despite the various attempts to alleviate the syntax with higher-level constructs like in PSL (Property Specification Language) [16].

Closing the gap between the requirements level and ESL is even more important for safety-critical systems for which we must ensure that what is verified is actually what was intended by the requirements. Indeed, making a formal verification of some model that is neither the deployed code (or circuit) nor the original requirements would be useless. A seamless methodology from the requirements to the code synthesis is only possible if (1) the requirements language has a formal semantics that can be maintained through all the refinement steps, we propose to rely on MARTE/CCSL for that, (2) the syntax is simple enough to be widely accepted by all the engineers involved in the process, that is why we propose to rely on UML and extensions rather than on ad-hoc formalism or structured grammars.

Our approach (see Fig. 1) contributes to the trend to build a Formal Specification Level as an intermediate level from natural-language requirements to code synthesis. However it finds its specificity by three characteristics that are not, to the best of our knowledge, used jointly in previous approaches. (1) A set of pre-defined primitive domain-specific property patterns, (2) A graphical UML/ MARTE formalism to capture the properties. Rather than having to rely on natural-language, the semantics of these graphical properties is given by a MARTE/CCSL specification, (3) Logical polychronous time [17] as a central powerful abstraction to capture both causal and temporal constraints. While CCSL gives the syntax to build these specifications, TimeSquare [18] can be used to execute the model, generate code and conduct verification [19]. Having executable requirement models helps reduce the risks of misinterpretation.

Our initial attempts were on the natural representation of UML diagrams to address the system requirements [20]. This paper proposes a complete framework to interpret the UML diagrams in a natural way and provide tools to transform those graphical representations into observers. Section II discusses the related work and their differences with the proposed approach. Popular LTL based property specification approaches like Property Sequence Charts (PSCs), TILCO-X, Metric Temporal Logic (MTL), Drag and Drop PSL (DDPSL) are discussed and compared to our proposed framework. Research work from the domain of runtime verification community is discussed next which mainly targets LTL, MTL and generation of efficient observers. Moreover, a comparison is made to the Formal Specification Level (FSL) in terms of their area of focus, utility, similarities and differences.

Section III in the first part provides the state of the art about UML artifacts while the latter part serves as a partial contribution in itself by proposing a UML profile extension targeting temporal patterns. The section first introduces UML state machine and sequence diagrams mentioning the features they lack. UML itself lacks the notion of time which is essentially required in modeling temporal patterns. So we introduce next the MARTE profile which provides desired timings concepts. The combined features of UML and MARTE provide a working ground for developing timed-models but they still lack the specialized features to model graphical temporal patterns. So in the last part of this section, we introduce the *Observation profile* which is designed to target the expressiveness of graphical temporal properties. This profile resides on top of the UML/MARTE to provide a structure for building some predefined patterns.

Section IV presents the proposed framework by specifying and formally defining these temporal patterns. While describing the behavior of a system we normally come across two things: states and events. The temporal patterns target both types of behavioral features. They use state machine diagrams to model generic temporal patterns representing relations between various states of a system. Sequence diagrams are used to model relations between states and individual events as they already have the valuable concepts of message passing, state invariant etc. Broadly these patterns are divided into state-state and state-event relations discussed one-by-one in detail. Then the section V applies these temporal patterns into use on a traffic light controller case study. The result of simulating observers in the design is shared at the end of the section. Once these patterns are established, we present their formal semantics in section VI. Selected temporal patterns from this framework are presented in CCSL. These CCSL relations can be

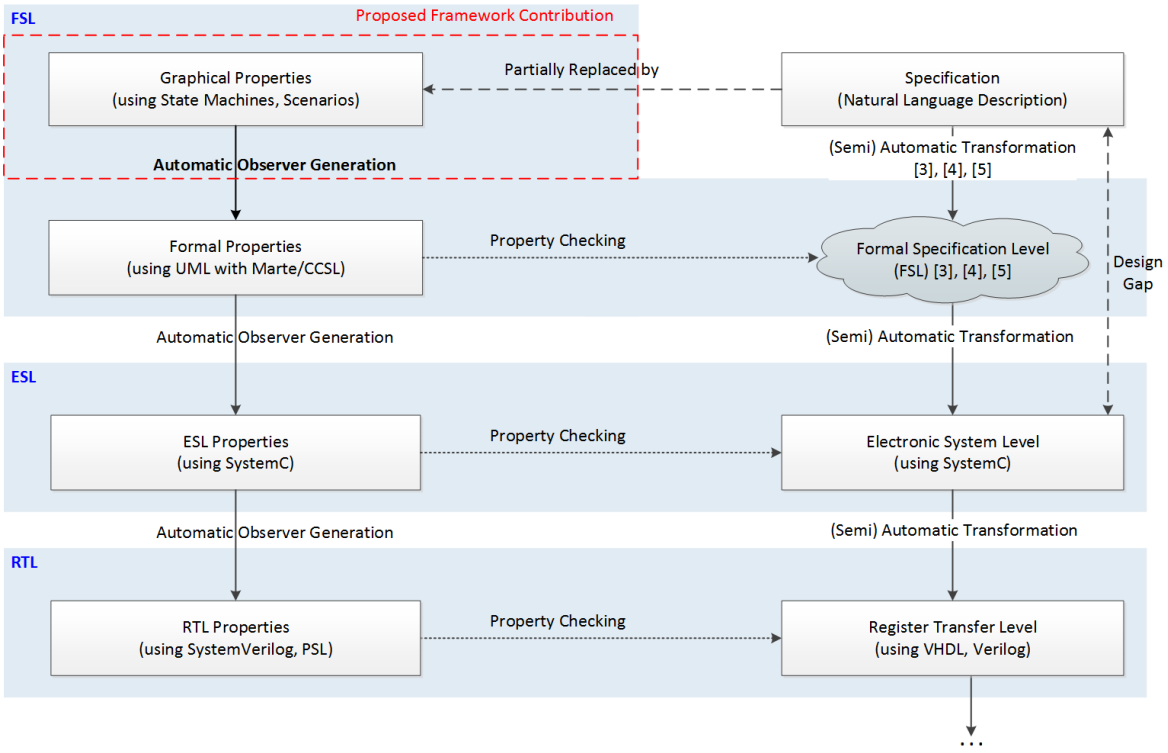


Fig. 1: Proposed Design flow

converted using TimeSquare tool into Verilog observers for direct hardware validation of the modules. Further, this section presents a more direct approach in the form of automaton diagram of temporal patterns. These automata are then used later to generate the desired Verilog code of the temporal pattern. Section VII discusses the tool implementation for the presented framework. It introduces the Eclipse plugin *TemPAC* (Temporal Pattern Analyzer and Code Generator) developed in Java EMF. It operates on the UML state machine and sequence diagrams to generate the CCSL or Verilog code from the modeled graphical temporal patterns. Finally section VIII concludes the paper with a glimpse over the future possibilities.

2 Related Work

Various efforts have been made over the past two decades to bridge the gap between natural language and temporal logic. Initially property specification patterns [21] were proposed in the form of a library of predefined LTL formulae from where the users can pick their desired pattern to express the behavior. Later other works proposed the specification of temporal properties through graphical formalism [22], [23], [24], and [25]. There have also been several attempts to encode temporal logics formula as UML diagrams [26, 27].

Property Sequence Charts (PSC) [28], [29] are presented as an extension to UML sequence diagrams to model well-known property specification patterns. Originally PSCs focused on the representation of order of events and lacked the support for the timed properties. But the later extension in the form of Timed PSC [26], [27] support the specification of timing requirements. PSCs mainly target LTL properties. The domain of expressiveness of CCSL is different from LTL and LTL-based languages, like PSL. CCSL can express different kind of properties than LTL [30]. Here we are interested to express properties on logical clocks for which LTL is not an obvious choice. Also LTL is not meant to express physical time properties, for which, this framework prefer the use of CCSL. Moreover, PSCs do not benefit from the popular embedded systems modeling and analysis profiles like MARTE. Rather than encoding formula with UML extensions,

we propose to reuse UML/MARTE constructs to build a set of pre-defined property patterns pertinent for the domain addressed.

The work presented by Bellini and his colleagues is a review of the state of the art for real-time specification patterns, organizing them in a unified way. It presents a logic language, TILCO-X, which can be used to specify temporal constraints on intervals, again based on LTL. The work of Konrad et al. [31] is to express real-time properties with a facility to denote time-based constraints in real-time temporal logics MTL (Metric Temporal Logic) extending the LTL. Finally, another research work, DDPSL (Drag and Drop PSL) [32], presents a template library defining PSL formal properties using logical and temporal operators.

The research domain of *runtime verification* [33,34] relies on lightweight formal verification techniques to check the correctness of the behavior of a system. Most works on such online monitoring algorithms focus on the LTL, CTL, MTL for expressing the temporal constraints [35,36] where high expertise is required to correctly capture the properties to be verified. Moreover, specialized specification formalisms that fit the desired application domains for runtime verification are usually based on live sequence charts (LSCs) or on MSCs [15]. Another research work to mention here is about the generation of controller synthesis from CCSL specifications [37]. Mostly the main focus of runtime verification community is on the generation of efficient observers for online monitoring whereas our proposed framework targets the integration of observers in a complete work-flow. The focus here is on the presenting a natural way to model temporal behavior.

Another aspect of the related work is the advent of Formal Specification Level (FSL), still an informal level of representation. The focus of the FSL approach is to transform natural language descriptions directly into models bridging the design gap (shown on the right in Figure 1). On the other hand, our proposed framework targets the graphical representation of temporal properties (red box on the left of Figure 1) replacing the need for textual specification of the system. *Natural language front-end* is a general trend to allow for a syntax-directed translation of concrete pattern instances to formulae of a temporal logic of choice, also used in [31] and [38]. Our framework approach is different from the FSL approach as we target the verification and validation of a subset of behavior rather than the complete system. Design engineers are usually well acquainted to UML diagrams (both state machine and interaction) and any graphical alternative to complex CCSL or LTL/CTL notations is more likely to get wide acceptance. Moreover, the use of MARTE profile allows to reuse the concepts of time and clocks to model timing requirements of embedded systems.

3 UML/Marte Semantics and Observation Profile

This paper proposes a framework to interpret the UML diagrams in a natural way. Hence the first sub-section introduces state of the art about the UML artifacts we consider: state machines and sequence diagrams. UML itself lacks the notion of time which is essentially required in modeling temporal patterns. So selected features from the MARTE time model are explained in the second sub-section which are used in the framework to facilitate semantically sound representation of time.

The combined features of UML and MARTE provide a working ground for developing timed-models but they still lack the specialized features to model graphical temporal patterns. The last part of this section serves as a partial contribution in itself by proposing a UML profile extension targeting temporal patterns. We introduce the *Observation profile* which is designed to target the expressiveness of graphical temporal properties. This profile resides on top of the UML/MARTE to provide a structure for building some predefined patterns.

3.1 UML State of the Art

UML *state machine diagrams* [39] provide a standardized way to model functional behavior of state-based systems. They provide behavior to an instance of a class (or object). Each state machine diagram basically consists of states an object can occupy and the transitions which make the object change from one state to another according to a set of well defined rules. Transitions are marked by guard conditions and an optional action. Formally a UML state machine can be defined by the 5-tuple:

$$StateMachine = \langle S, R, top, container, \mathcal{T} \rangle$$

where

- S is a finite set of states consisting of simple states S_{simple} , composite states $S_{composite}$, final states S_{final}, initial pseudo-states $S_{initial}$ and choice pseudo-states S_{choice}.
- R is a finite set of regions (disjoint from S).
- $top \in R$ is the unique top region.
- $container : (S \cup R \backslash \{top\}) \rightarrow (S \cup R)$ describes the state hierarchy of the state machine, and
- $\mathcal{T}$ is a finite set of transitions

Among the set of state machine elements defined, only a small subset is used by the presented framework to represent graphical properties by giving specific semantics to that chosen subset. The framework specifies S as a set of finite states consisting of simple states S_{simple} , final states S_{final} and choice pseudo-states S_{choice} while the other states (like initial pseudo-states $S_{initial}$) are not used at all. From the standard set of state machine elements, only the top region is used while all other set of regions like in state hierarchy are not considered. The framework on considers $top \in R$ as the unique top region and does not consider any other region in R. Finally, $\mathcal{T}$ is considered as a finite set of valid transitions.

UML *sequence diagrams* [39] allow describing interactions between system objects and actors of the environment. A sequence diagram describes a specific interaction in terms of the set of participating objects and a sequence of messages they exchange as they unfold over time to effect the desired operation. Sequence diagrams represent a popular notation to specify scenarios of the activities in the form of intuitive graphical layout. They show the objects, their lifelines, and messages exchanged between the senders and the receivers.

A sequence diagram specifies only a fragment of system behavior and the complete system behavior can be expressed by a set of sequence diagrams to specify all possible interactions during the object life cycle. It is useful especially for specifying systems with time-dependent functions such as real-time applications, and for modeling complex scenarios where time dependency plays an important role. Sequence diagrams consist of objects, events, messages and operations. *Objects* represent observable properties of their class(es). Object existence is depicted by an object box and its 'life-line'. A life-line is a vertical line that shows the existence of an object over a given period of time. An *event* is a specification of a significant occurrence having time and space existence. A *message* is a specification of a communication that conveys information among objects, or an object and its environment. The formal abstract syntax of sequence diagram is given next, which is used to construct well-formed sequence diagrams.

$$
\begin{aligned}
&SequenceDiagram ::= \\
&\qquad sdname \stackrel{\text{def}}{=} CombinedFramgment \\
&CombinedFragment ::= Interaction| \\
&\quad CombinedFragment; CombinedFragment| \\
&\quad \mathbf{opt}(Cond, CombinedFragment)| \\
&\quad \mathbf{alt}(Cond, CombinedFragment, CombinedFragment)| \\
&\quad \mathbf{loop}(Cond, CombinedFragment)| \\
&\quad \mathbf{loop}(1, n, CombinedFragment) \\
&Interaction ::= skip|\mathbf{ref}(sdname)|Message \\
&\qquad |Message \stackrel{\text{def}}{=} CombinedFragment \\
&Message ::= (Sender, A, Receiver, MethodCall) \\
&Cond ::= booleanexpersion \\
&Sender ::= objectname : CN \\
&Receiver ::= objectname : CN \\
&CN ::= classname
\end{aligned}
$$

$$A ::= associationname$$
$$MethodCall ::= method(para)$$

where $\stackrel{\text{def}}{=}$ expresses the left side message method call will invoke the messages inside of the brace body {.}. This abstract syntax helps us define the message as a tuple:

$msg = (s : S, A, r : R, m(para))$,

where s is the sender object of the message with class type S, r is the receiver object of the message with class type R, A is an association between classes S and R, and $m(para)$ is a method call from sender object s to receiver object r with the specified parameter $para$. From the abstract syntax, the sequence diagram can also be defined as:

$\Delta = (ObjectSet, MessageSet)$

in which $ObjectSet$ is the set of objects which participate in the sequence diagram and $MessageSet$ consists of all the messages in the diagram numbered in a sequence demonstrating the possible execution order as well as the implementation relationships among messages.

Just like the state machine diagrams, the proposed framework focuses on a subset of sequence diagram elements. Amongst the combined fragment elements, this work only uses the *consider* fragment and interaction diagrams are not used in a hierarchical fashion. Moreover, *StateInvariant* are used in the scenarios to represent *ActivationState* similar to the state machines, showing the triggering of specific state event.

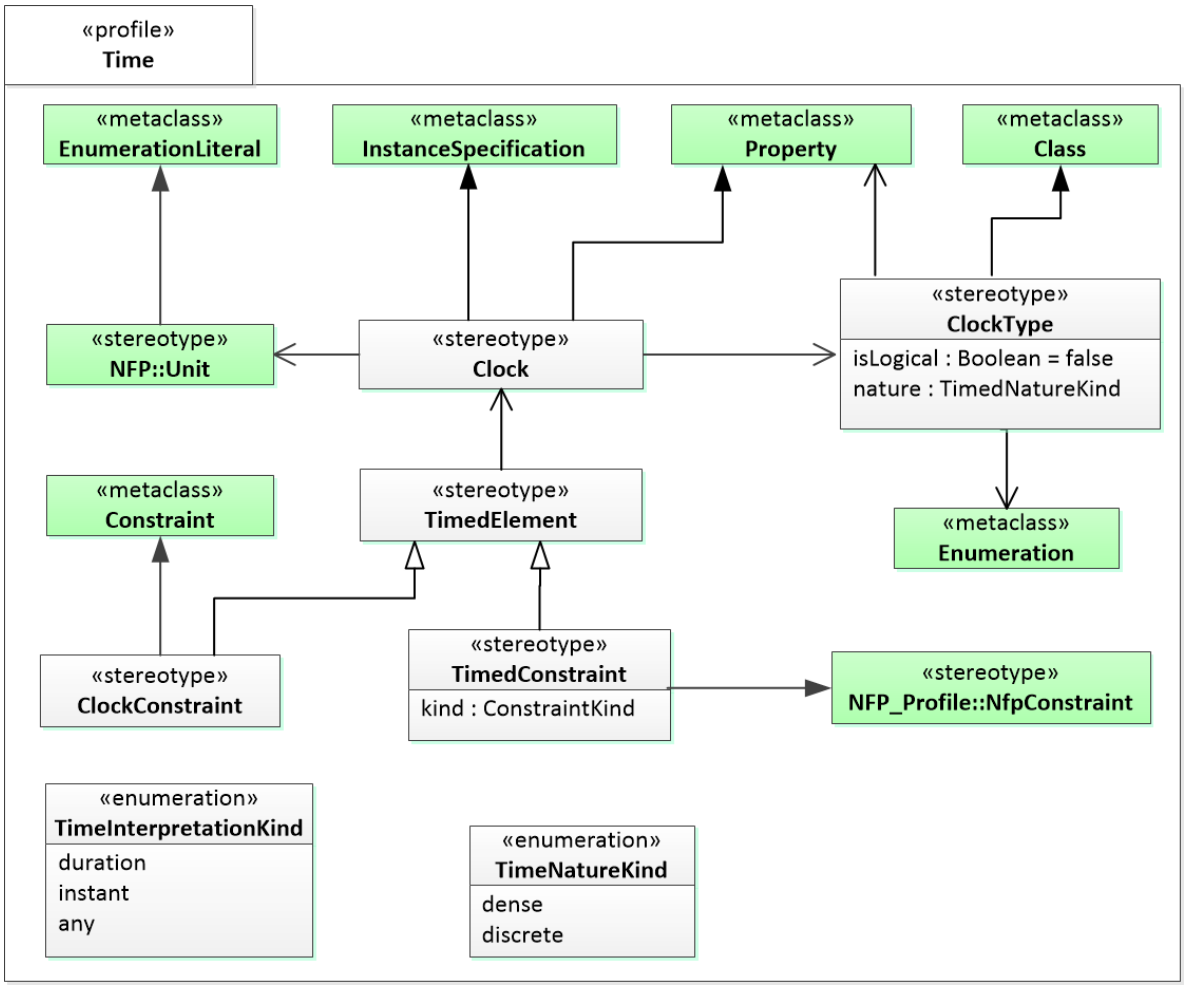


Fig. 2: Excerpt of MARTE Time Sub-profile

3.2 MARTE Time Model and CCSL

The proposed framework uses concepts of clocks and time for which MARTE time model [8,40] is utilized. MARTE time model provides a sufficiently expressive structure to represent time requirements of embedded systems. In MARTE, time can be physical viewed as dense or discretized, but it can also be logical related to user-defined clocks. Time may even be multiform, allowing different times to progress in a non-uniform fashion, and possibly independently to any (direct) reference to physical time. MARTE time model is a set of logical clocks and each clock can be represented by $\langle \mathcal{I}, \prec \rangle$, where $\mathcal{I}$ represents the set of instants and $\prec$ is the binary relation on $\mathcal{I}$.

Figure 2 presents a simplified view of MARTE Time sub-profile. The green elements are not part of the time sub-profile. At the heart of the profile, the stereotype *ClockType* extends the metaclass *Class* while the stereotype *Clock* extends metaclasses *InstanceSpecification* and *Property*. Clocks can appear in structural diagrams (like SysML block definition, internal block definition, or UML composite structure) to represent a family of possible behaviors. This clock association gives the ability to the model elements identifying precisely instants or duration. MARTE introduces *ClockConstraint* stereotype extending the metaclass *Constraint* through which a MARTE timed system can be specified. *TimedConstraint* is a constraint imposed on the occurrence of an event or on the duration of some execution, or even on the temporal distance between two events. The presented framework uses the *TimedConstraint* on the sequence diagram *DurationConstraint* elements, discussed further in the text later.

The CCSL (Clock Constraint Specification Language) [10] is a declarative language annexed to the specification of the MARTE UML profile. It is used to specify constraints imposed on the clocks consisting of at least one clock relation. A clock relation relates two clock specifications. A clock specification can be either a simple reference to a clock or a clock expression. A clock expression refers to one or more clock specifications and possibly to additional operands. The clock relations can be classified as synchronous, asynchronous, or a combination of both. There are three basic clock relations in CCSL: precedence ($\boxed{\preccurlyeq}$), coincidence ($\boxed{\equiv}$), and exclusion ($\boxed{\#}$). Two more relations, the strict precedence ($\boxed{\prec}$) is derived from the precedence relation while the subclocking ($\boxed{\subset}$) relation is a one-to-one coincidence between one clock and a subset of events of another clock. Subclocking constraint ($\boxed{\subset}$) is an example of synchronous clock constraint based-on coincidence. Each instant of the subclock must coincide with one instant of the superclock in an order-preserving fashion. The exclusion constraint ($\boxed{\#}$) states that the instants of the two clocks never occur at the same time. Non-strict precede constraint ($\boxed{\preccurlyeq}$) is an example of asynchronous clock constraint based-on precedence. Given the relation a $\boxed{\preccurlyeq}$ b, for all natural numbers k, the k^{th} instant of 'a' precedes or is coincident with the k^{th} instant of 'b' ($\forall \in \mathbb{N}, a[k] \preccurlyeq b[k]$). Mixed clock constraints combine coincidence and precedence relations. An example is defer constraint ($\rightsquigarrow$) which enforces delayed coincidences. The expression $c = a(ns) \rightsquigarrow b$ (read as *a deferred b for ns*) imposes c to tick synchronously with the n^{th} tick of b following a tick of a. Another mixed clock constraint is the strict sampling $a \searrow b$ which defines a subclock of 'b' that ticks whenever clock 'a' has ticked at least once since the previous tick of b.

3.3 Observation Profile

The focus of the presented work is not the minimalist approach but rather the expressiveness of the property from the system designer's point-of-view. So when a property specifies occurrence of something at some point in time, then it is an event and it seems natural to represent such properties as logical clocks. But when the properties specify duration or interval (not a particular point in time), then the obvious choice of representation is state relations.

In the proposed framework, extended state machine diagrams are mainly used to represent the behavior patterns of a system. These diagrams provide a more natural and syntactically sound interpretation of graphical behavioral patterns of system states. Moreover, some of the system state/event relation patterns are mapped to sequence diagrams in this presented framework. State machines are used to model state-based relations. State machines lack some required features for which a profile is introduced, as shown in Figure 3. This profile presents *ObservationScenario* stereotype extending the UML state machine metaclass. Stereotypes *ActivationState* and *Duration* extends the state metaclass. Using the ObservationScenario stereotype, state-based system properties can be represented graphically in the

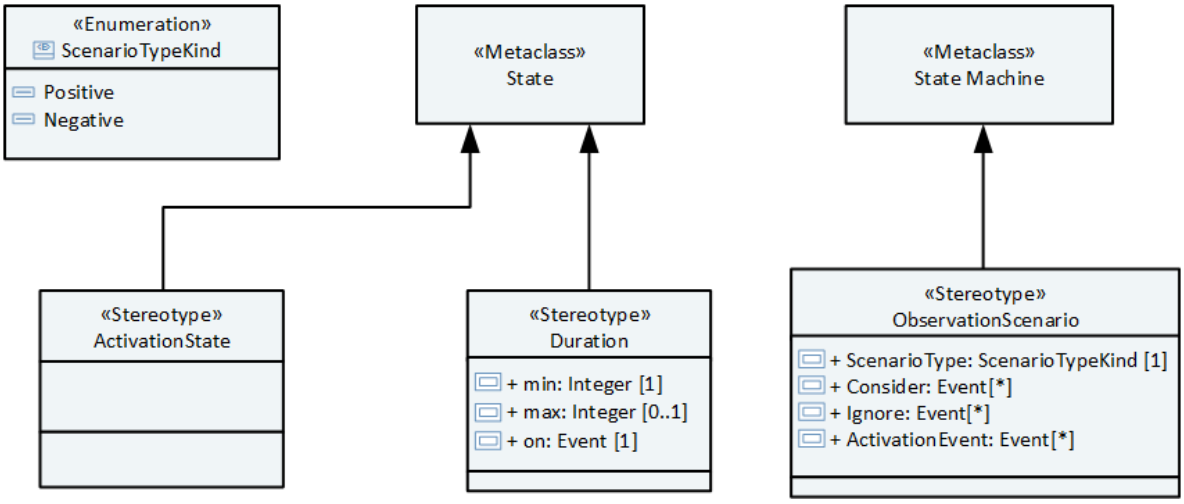


Fig. 3: Proposed Observation Profile

form of scenarios. Two basic scenarios are possible: negative and positive. These scenarios act as building blocks for more complex state patterns.

Positive scenarios model something that must happen under given conditions, as shown in Figure 4. *Consider* contains the collection of all the events that are relevant to this scenario. It is just like the sensitivity list in SystemC, Verilog or VHDL. If the list of relevant events is large, then the list of events that are not relevant maybe modeled using the *Ignore*. *ActivationState* stereotype is used to identify the state that activates this particular scenario. It is active whenever the system is in a specified condition. The positive scenario expects an event to occur whenever the considered state is active. Failure occurs if the event does not occur. The scenario terminates normally if the desired event occurs. *Negative scenarios* model something that must not happen under given conditions. So when the state machine is active, it checks for a particular trigger event that leads the system to a violation/error state shown in Figure 4. This type of properties can use model-checking to detect if the system under observation ever reaches an error state. The *Duration* stereotype is used to model the delay optionally in some case of temporal patterns. Such a stereotype is only required here for the state machines while the sequence diagrams utilize the existing features of MARTE TimedConstraint stereotype.

4 Proposed Temporal Patterns

The first major contribution of this proposed framework is to provide a set of reusable generic graphical temporal patterns. For identifying the temporal properties of systems, we started by considering several examples like the famous steam boiler case study [41, 42], railway interlocking system [43] and the traffic light controller case study [44]. Working on these diverse examples to model behavioral properties in UML, we noted that several temporal patterns were repeated across different examples. These patterns collected across various examples were refined with some inspiration taken from the Allen's algebra targeting intervals [45]. This practice gave us a valuable collection of generic patterns divided into three major categories of behavioral relations that may exist in a system: state-state relations, state-event relations, and event-event relations. Researchers have already shown that constraints specified in CCSL are capable of modeling logical event-event temporal relations [40, 46]. These CCSL constraints can be represented graphically using SysML/MARTE models [47–49]. Temporal patterns for the other two categories of relations are:

State-State Relations: precedes, triggers, contains, starts, finishes, implies, forbids, and excludes.

State-Event Relations: excludes, triggers, forbids, contains and terminates.

Next subsections discuss selected few of these temporal patterns. A detailed list of these patterns with their syntax and semantics is available online [50].

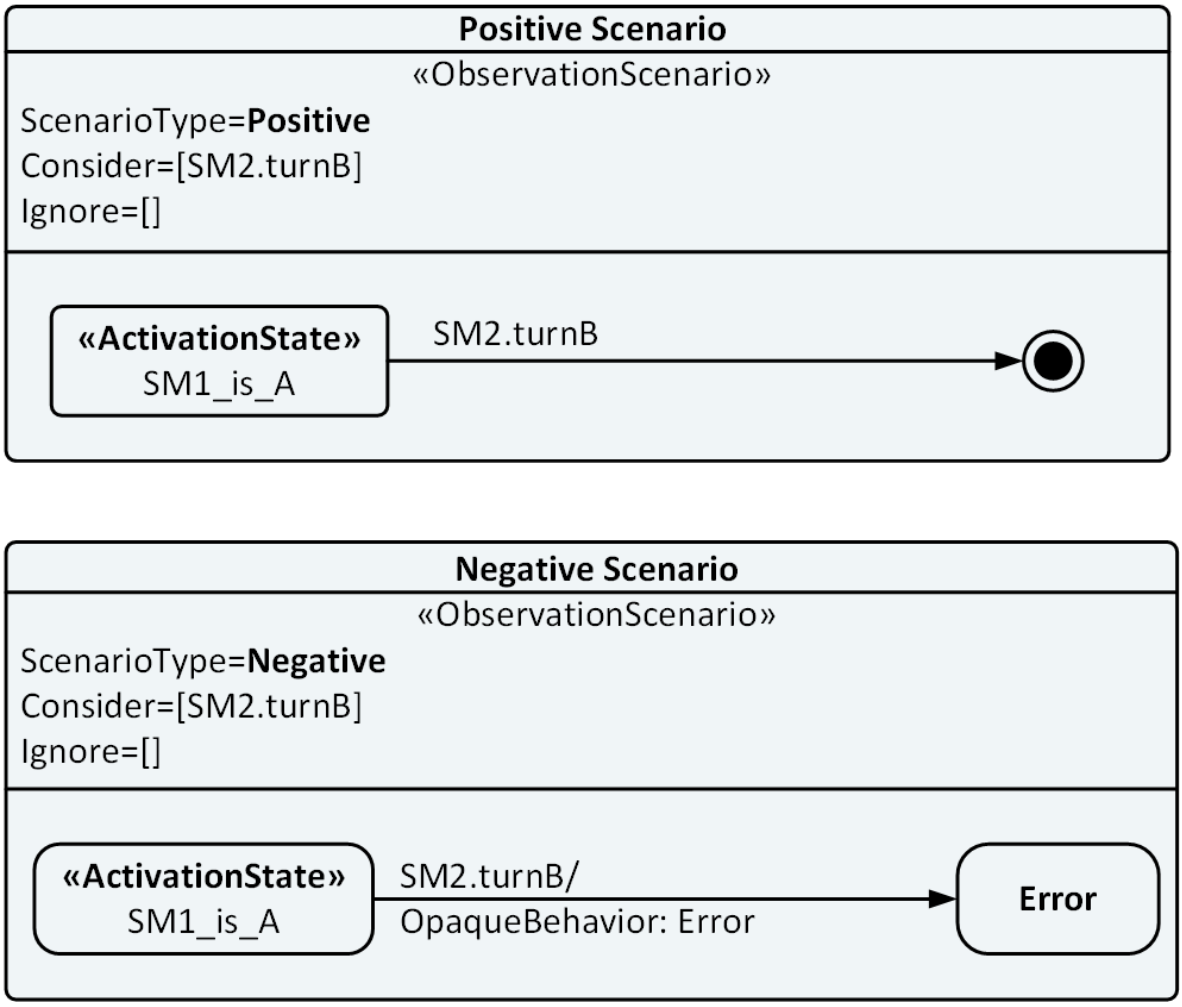


Fig. 4: Positive and Negative Scenarios

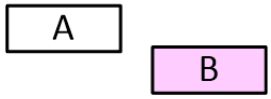
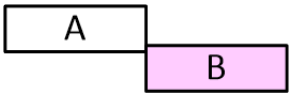
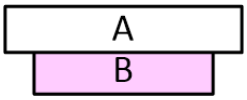
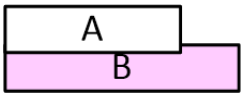
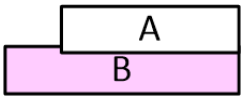
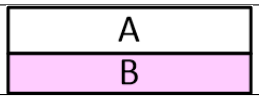
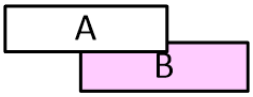
4.1 State-State Relations

Presented framework including the two basic scenarios can be used to formally model relations between two different states of a system. Allen's algebra [45] for intervals provides a base defining thirteen distinct, exhaustive and qualitative relations of two time intervals, depicted in Table 1. From the comparative analysis of these relations, six primitive relations are extracted that can be applied to the state-based systems. Further two 'negation' relations are added, based on their usage and importance in the examples, to complete the set. The overlapping of states is not particularly interesting relation to dedicate a pattern for. But it can easily be modeled indirectly using the state-event relation 'A contains b_s' where b_s is the start event of state B.

Semantically a state can be considered similar to an interval. We use the nomenclature of using capital letters (A,B,...) to denote states and small letters (a,b,...) for events/clocks. Dot notation like SM1.turnA is also used throughout the text to show the specific events. Given a strict partial ordering $\mathbb{S} = \langle S, < \rangle$, a state in $\mathbb{S}$ is a pair $[a_s, a_f]$ such that $a_s, a_f \in$ S and $a_s < a_f$. Where a_s is the start and a_f is the end of the state interval. An event or point e belongs to a state interval $[a_s, a_f]$ if $a_s \leqslant e \leqslant a_f$ (both ends included).

Precedence is an important state property where the relation 'A precedes B' means the state A comes before the state B. It includes a delay/deadline clause to explicitly specify the duration between the termination of state A and the start of state B. This delayed version also be equated to the triggers

Table 1: Allen's Algebra and Proposed Relations

No.	Graphic	Allen's Algebra for Intervals	State-State Relation	Symbol
1		A precedes B	A precedes B	$\boxed{\preccurlyeq}$
2		B preceded-by A		
3		A meets B	A triggers B	$\boxed{\vdash}$
4		B met-by A		
5		A contains B	A contains B	$\boxed{\supseteq}$
6		B during A		
7		A starts B	A starts B	$\boxed{\vdash}$
8		B started-by A		
9		A finishes B	A finishes B	$\boxed{\dashv}$
10		B finished-by A		
11		A equals B	A implies B	$\boxed{\Rightarrow}$
12		A overlaps B	See the text	
13		B overlapped-by A		
			A forbids B	$\boxed{\neg}$
			A excludes B	$\boxed{\#}$

state-event relation (e triggers A after [m,n] on clk). The unit of the duration is dependent on the level of abstraction that is target of the graphical specification.i.e, it can be physical clock, loosely timed clock, or logical clock. Deadline defers the evaluation of state A until some number of ticks of clk (or any other event) occur. The number of ticks of clk considered are dependent on the two parameter natural numbers min and max evaluated as:

- [0, n] means 'before n' ticks of clk
- [m, 0] means 'after m' ticks of clk
- [m, m] means 'exactly m' ticks of clk

Mathematically, given a partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), a constant n and a clock clk, the equation

$$A \boxed{\preccurlyeq} B \quad by\ [m, n]\ on\ clk$$

means $a_f \leqslant b_s$ and b_s occurs within the duration $a_f + \Delta$, where Δ is between m and n ticks of event clk. The last tick of clk coincides with the start the state B (i.e, b_s). Graphically, precedence is based on positive scenarios shown in Figure 5. The first state (State_is_A) is an activation state (shown by the stereotype) while the second state has the duration stereotype applied to specify the interval. The observation scenario gets active when the state machine SM1 is in A state. It then checks for the state exit (turnNotA) and expects the other state machine SM2 to be in state B within the specified time duration. If this behavior occurs as desired, then the scenario goes dormant till the next state activation occurs.

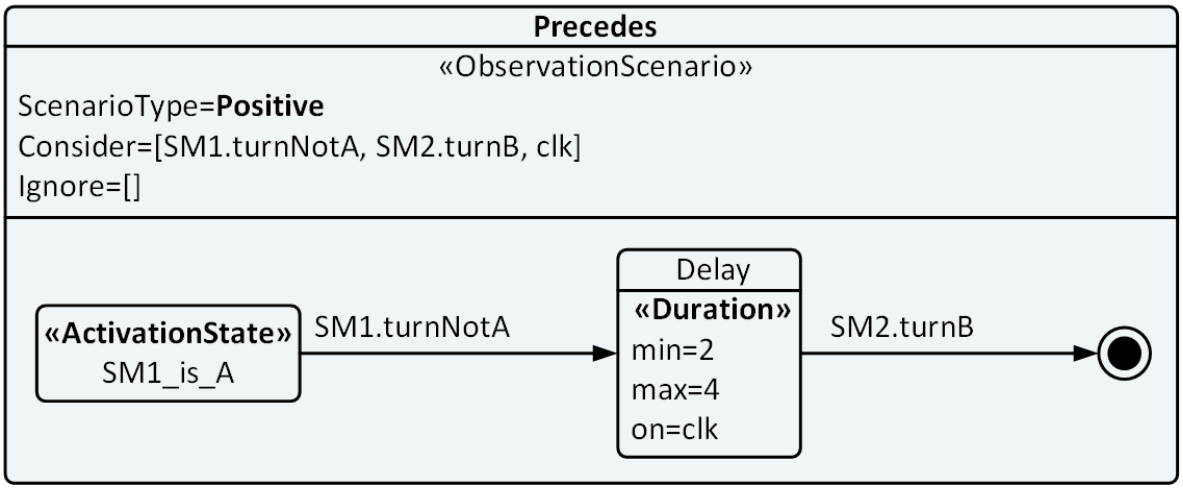


Fig. 5: A precedes B by [2, 4] on clk

Forbiddance is a negation property. Relation 'A forbids B' bars B to occur after state A occurs. It has another slightly different operator that works with events (e forbids A), discussed later on. Hence mathematically, given a partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the equation $A \boxed{\neg} B$ means $b_s \neq a_f$. Its graphical temporal pattern is shown in Figure 6. Scenario activates whenever SM1 is in state A and on exiting this state, the SM2 is expected to be not in state B (else violation occurs).

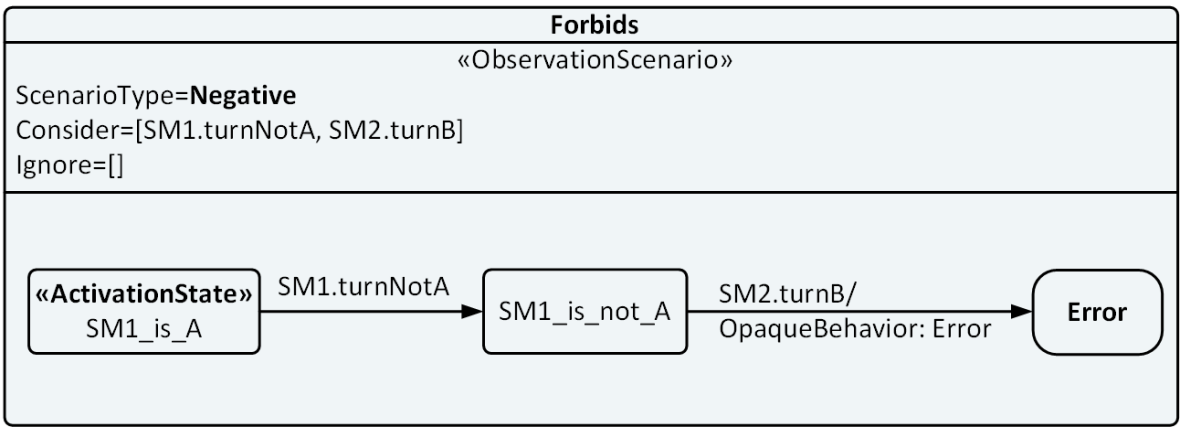


Fig. 6: A forbids B

Exclusion between two states restricts them to occur at the same time. Mathematically, given a partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the relation 'A excludes B' means that $b_f < a_s$ and $a_f < b_s$ for all instances of A and B. This relation can be decomposed into two basic state-event exclusion relations (shown without boxed symbols).

$$A \# bs \; and \;\; B \# as$$

Graphically, this temporal pattern is derived from the exclusion relation of state and event (discussed in the next sub-section). Two negative scenarios are used to model this behavior as shown in Figure 7. So during the particular state A for SM1, the event turnB is expected not to occur and vice versa for the other case.

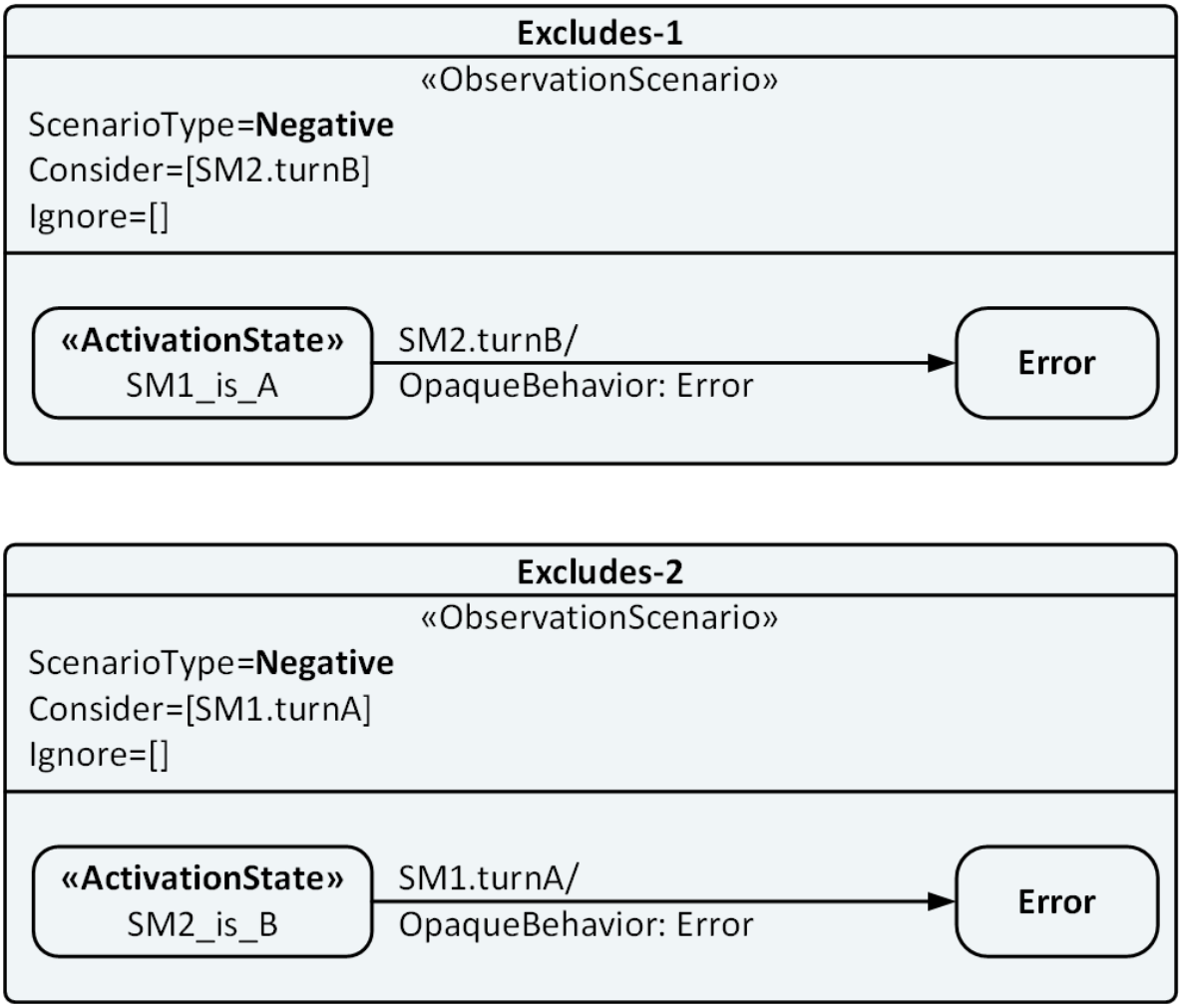


Fig. 7: A excludes B

4.2 State-Event Relations

The relations between the system states and events can mostly be modeled using the UML sequence diagrams which suits modeling flow of events. The concept of state invariant is used to represent the system activation states. Moreover the sequence diagrams already have the consider/ignore in the form of combined fragments which were introduced earlier in the state machines using the *Observation* profile. Based on the use, we identify four state-event relations.

The **excludes** relation *state A excludes event e* is a bijective relation. Mathematically, given a partial ordering $\mathbb{S}$ having the state A ($[a_s, a_f]$) and a clock e, it can be expressed as $A \# e$ where $e \notin [a_s, a_f]$. It implies either $e < a_s \quad or \quad a_f < e$. Graphically it is modeled using a negative scenario, as shown in Figure 8. Here the SM1 while in state A is causes error on event e.

The **triggers** relation is similar to triggers and starts relations for states. The relation *event e triggers state A* can be expressed mathematically, given a partial ordering $\mathbb{S}$ having the state A ($[a_s, a_f]$), an event e, a constant n and a clock clk, as

$$e \vDash A \quad after\ [m, n]\ on\ clk$$

It means a_s occurs within the duration $e + \Delta$, where Δ is between m and n ticks of event clk. Graphically, sequence diagram is used to model such relations as shown in Figure 9. The two lifelines represent the

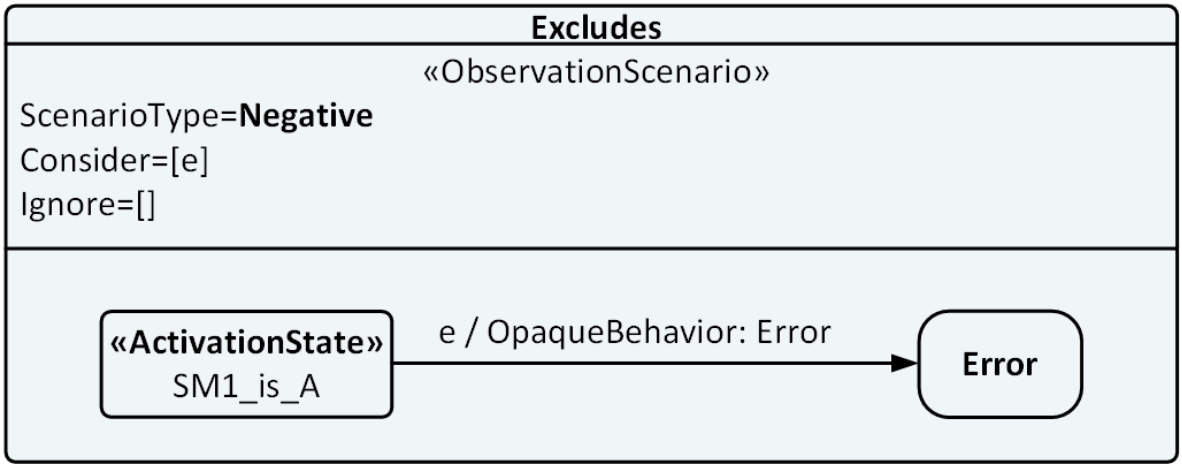


Fig. 8: A excludes e

system under test and the observer. The *consider* combined fragment maintains the list of participating events for the temporal pattern just like ObservationScenario did for the state machines. *StateInvariants* are used in sequence diagrams to represent activation states. The state invariant 'SM1_is_A'represents the conditions when the state machine SM1 is in state A. *Duration constraint* element is used to specify the required delay. MARTE stereotype *TimedConstraint* is used to further specify the unit of interval and the associated clock.

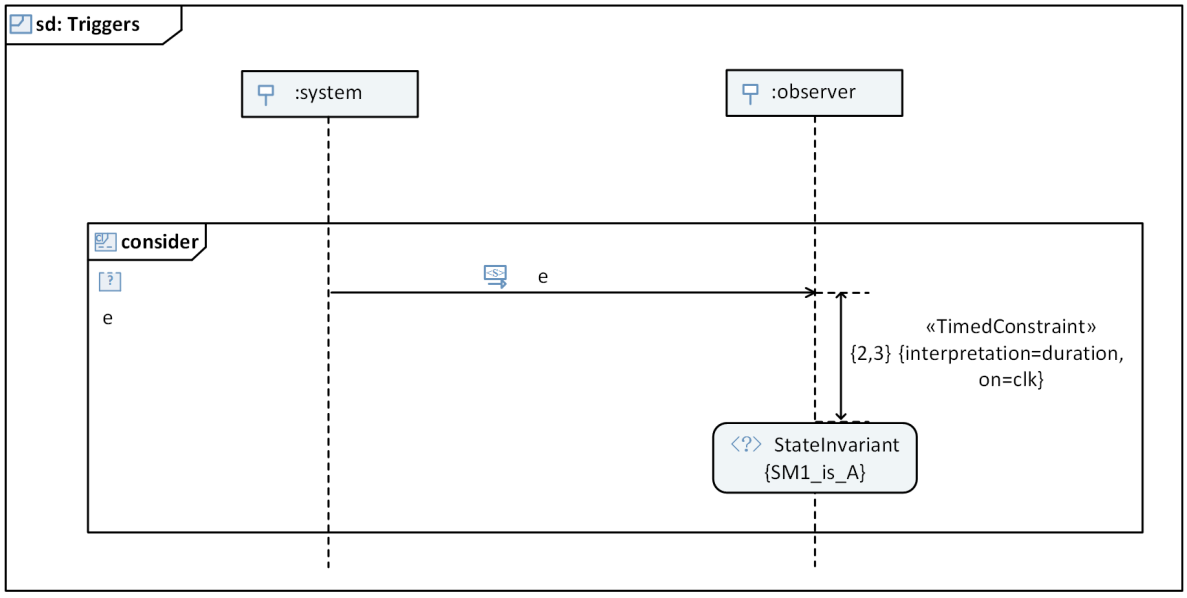


Fig. 9: e triggers A after [2,3] on clk

The **forbids** relation is similar to forbids relation for states which is implemented using state machines but this forbids relation for events is implemented using sequence diagrams (it only has one state to consider). Event e forbids state A implies A must not occur after the event e triggers. Hence given a partial ordering $\mathbb{S}$ having the state A ($[a_s, a_f]$), an event e, the relation is expressed mathematically as $e \neg A$ which means $e \neq a_s$. As this relation involves an event, the graphical temporal pattern is best expressed using sequence diagram, as shown in Figure 10. Here in the forbids relation, the state invariant

'SM1_is_not_A'represents the conditions when the state machine SM1 is not (either not entered or already left) in state A .

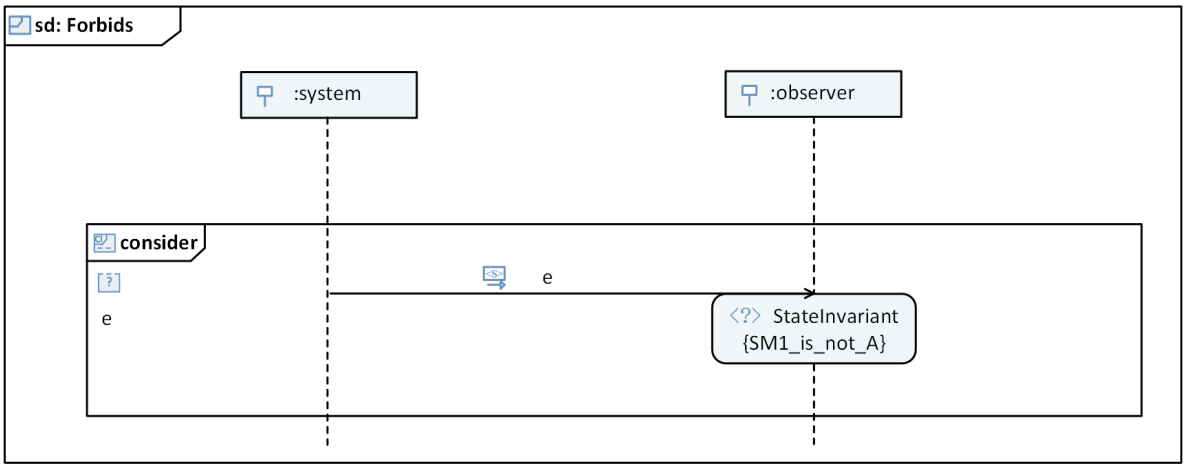


Fig. 10: e forbids A

The **terminates** relation is similar to finishes relation for states. The relation *event e terminates state A* can be expressed mathematically, given a partial ordering $\mathbb{S}$ having the state A ($[a_s, a_f]$), an event e, a constant n and a clock clk, as

$$e \dashv A \quad after\ [m, n]\ on\ clk$$

It means a_f occurs within the duration $e + \Delta$, where Δ is between m and n ticks of event clk. Graphically, it is implemented using the positive scenario state machine, as shown in Figure 11. Here it is important to mention that why a state machine is used for an event-based scenario. Here the important point to note down is that our event e will only trigger when the state machine SM1 is in state A (only then it can terminate). So obviously and activation state is required which will then lead to trace desired event.

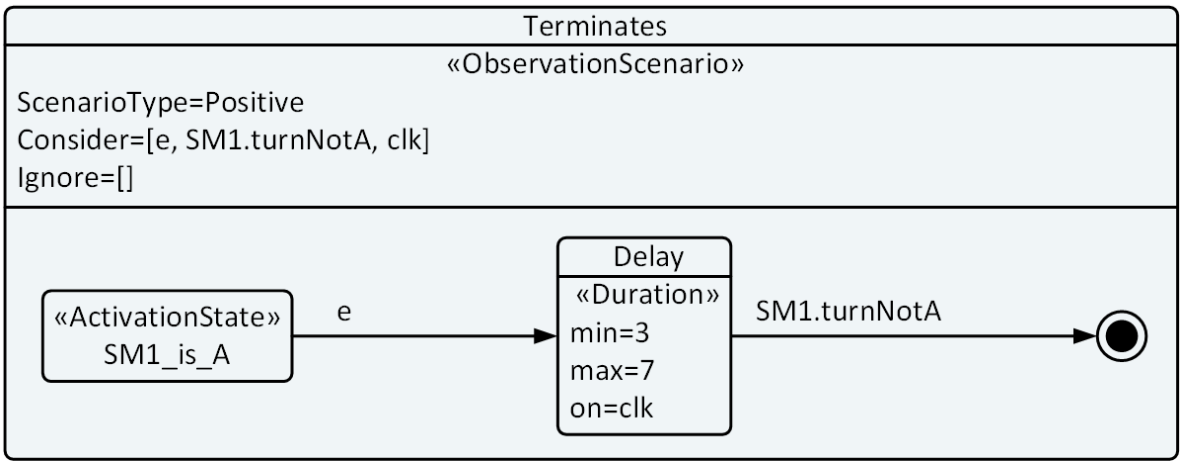


Fig. 11: e terminates A after [3,7] on clk

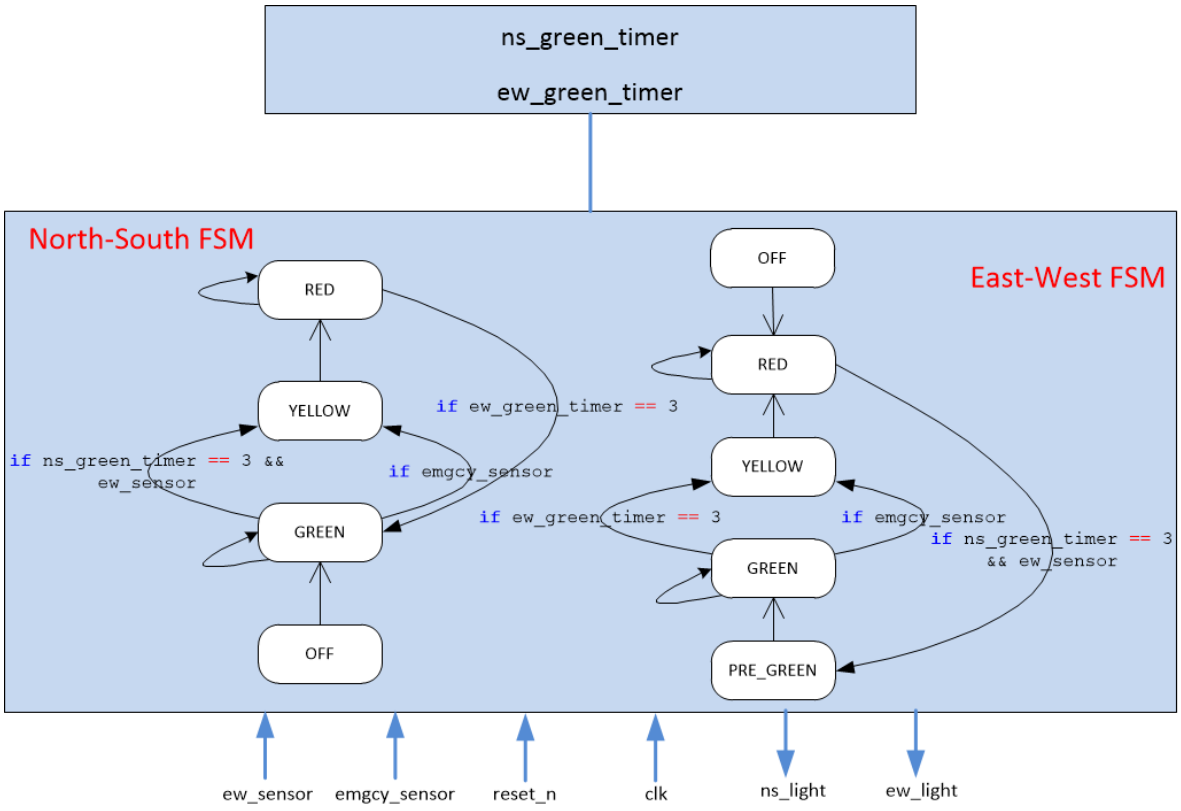


Fig. 12: Traffic Intersection Controller Module

5 Application of Temporal Patterns

The case study considered to demonstrate the approach is of the traffic light controller taken from the SystemVerilog Assertions Handbook [44]. It consists of a cross-road over North-South highway and the East-West farm road. There are sensors installed for the emergency vehicles and for the farm road traffic. Highway traffic is only interrupted if there is a vehicle detected by the farm road sensor. The architecture for the traffic light controller consists of two state machines, interface signals of the module and the timers, as shown in Figure 12. A few temporal verification properties of the design are discussed next.

Safety property, Never NS/EW lights both green simultaneously. This property is the exclusion of two states ns_light.GREEN and ew_light.GREEN. From our library of graphical temporal patterns, we consider two state-event excludes temporal patterns, as shown in Figure 13. Here if ns_light_is_GREEN is the activation state from SM1, then in the generic pattern we replace the event e with the start event of the opposite light (turnGREEN of ew_light).

State of lights at reset. This constraint requires that whenever reset occurs, the ns_light turns off. This property shows that ns_light.OFF is the consequence of reset. From our library of graphical properties, we use the implies operator for the relation, shown in Figure 14. The implies relation is like the excludes state-state relation as it is further composed of two state-state relations: *starts* and *finishes*. The starts relation guards the beginning of implies relation while the finishes guards the end of the implication relation. Both the relations are implemented using positive scenarios.

State of lights during emergency. This constraint requires that whenever the emergency sensor triggers, ns_light switches from GREEN to YELLOW to RED. The book uses the timing diagram to explain the intended timing relation of the constraint. Text further specifies at another place (chapter 7, SystemVerilog assertions handbook [44]);

The design also takes into account emergency vehicles that can activate an emergency sensor. When the emergency sensor is activated, then the North-South and East-West lights will turn RED*, and will stay* RED *for a minimum period of 3 cycles.*

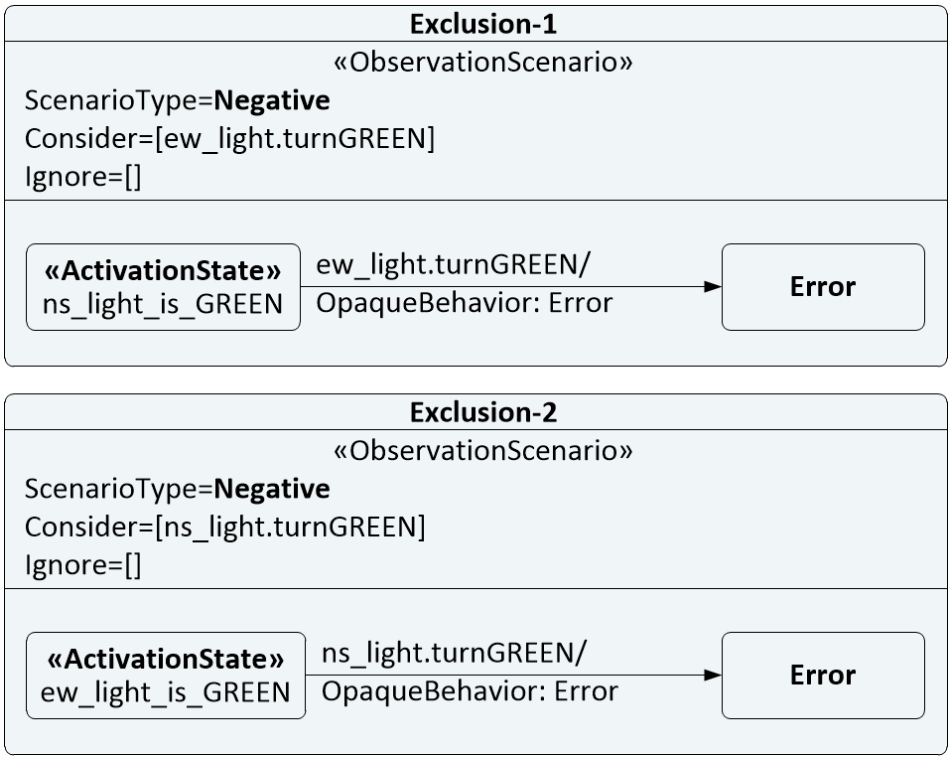


Fig. 13: ns_light=GREEN excludes ew_light=GREEN

Yet the SystemVerilog assertion for the constraint tests the ns_light equals RED after two cycles of the emergency. The YELLOW state is never tested. This vindicates our statement that the textual requirement specifications are usually ambiguous, not precise, bulky and the information is scattered. We can implement this property using the triggers relation for the event emgcy_sensor and the state ns_light_is_RED, as shown in Figure 15. A delay of exactly one clock cycle is shown using the duration constraint and the MARTE stereotype.

Safety, green to red is illegal. Need yellow. This constraint is another example of the difference between the textual specification and the constraint implemented as assertion. Though the YELLOW state is specified in the text but it is never tested in the assertion. Here the graphical approach is clear and precise in using the *precedes* relation for states without defining any delay, shown in Figure 16. Here the name of the state 'ns_light_is_not_YELLOW' is not required (as it is not an activation state), and any desired name can be used. This property here ensures YELLOW comes before RED. How much before that is not specified. To avoid cases like YELLOW ⇒ GREEN ⇒ RED, we can add another constraint *ns_light*=GREEN *precedes ns_light*=YELLOW. A little varying intent can also be implemented using the forbids relation *ns_light*=GREEN *forbids ns_light*=RED which seems to be the desired one for the text 'green to red is illegal'

5.1 Observers

Graphical interpretation of properties are implemented in the form of *observers*. Verification by observers is a technique widely used in property checking [51–53]. They check the programs for the property to hold or to detect anomalies. In the presented framework, each temporal pattern is finally transformed into a unique observer code for a specific abstraction level (like TLM, RTL). It proposes to create a library of verification components for each graphical temporal pattern. An observer provides implementation to the semantically sound graphical patterns. An observer consists of a set of input parameters, one for each activation state and event. A special violation output is there to flag any anomaly in the behavior.

One important thing to note here is that there is a gap between the way property is captured in Verilog or other low-level HDLs and what the system specification actually requires. So to build these graphical patterns we assure that everything is explicit. When these patterns speak about state, we have state information to model and when they speak about events then we have event information. The way these patterns work is by relying on *adapters* as a glue logic. These adapters convert the signal or group of signals from the system to states and events. The property patterns implemented in the framework use these events and states. So adapters come in-between the module under verification and the observers. They receive inputs in the form of design module interface signals and state values. From this they generate the appropriate logical clock outputs and state identifiers to be consumed by the observers. Figure 17 shows the integration of observer code in the verification environment. Every design language should have its own set of adapters e.g., if the design module is in VHDL, adapters written in VHDL should be used that will interact with the design and provide the appropriate inputs to the observer. For example, *ns_light_is_*GREEN is a signal that is true whenever the traffic light output ns_light is in GREEN state. In Verilog, the adapter code for the state is given next.

```
always @ (posedge clk) begin
    nslightGREEN = (ns_light == GREEN);
end
assign ns_light_is_GREEN = nslightGREEN;
```

The clock *ns_light.turn*GREEN is the rising edge of this state output and represents the change in the state. In SystemVerilog, we can implement this logic using the *$rose* or the *$past* operators.

```
ns_light_turnGREEN = $past(ns_light!=GREEN)
                     && ns_light==GREEN;
```

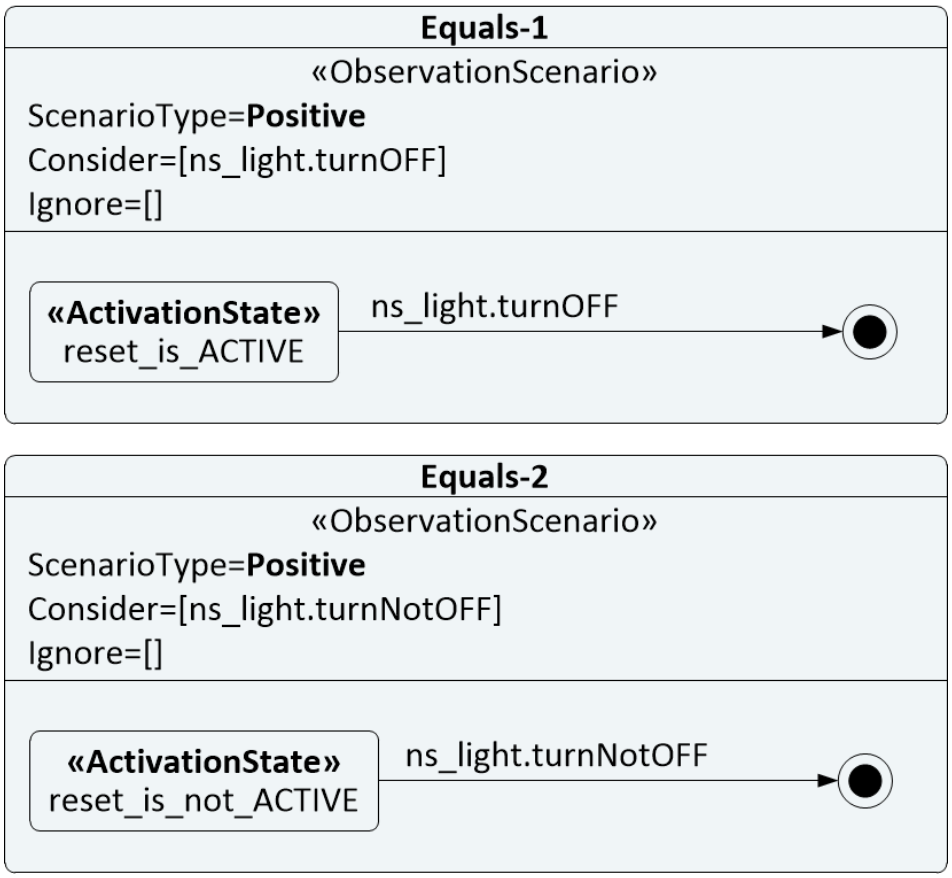


Fig. 14: reset=ACTIVE implies ns_light=OFF

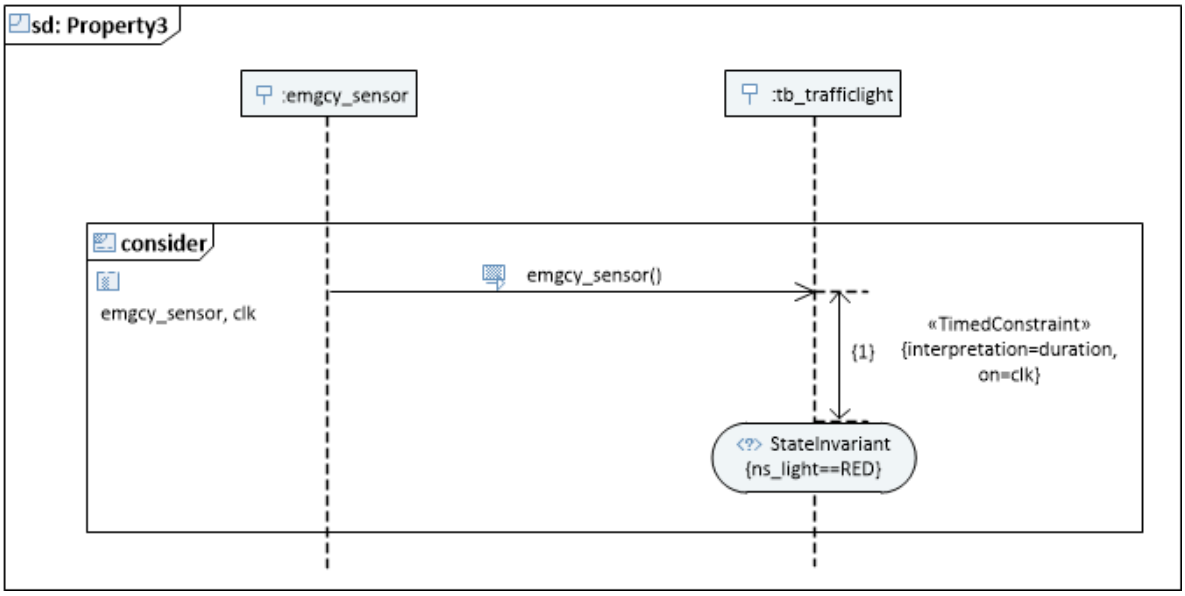


Fig. 15: emgcy_sensor triggers ns_light=OFF after [1,1] on clk

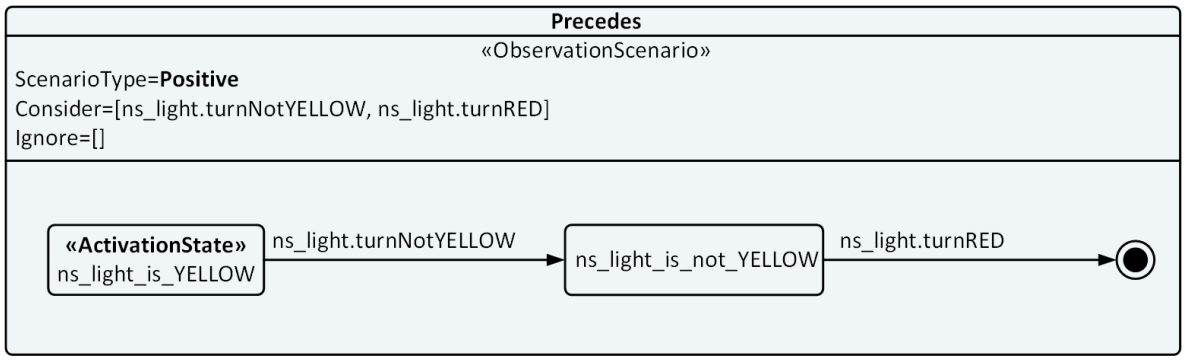


Fig. 16: ns_light=YELLOW precedes ns_light=RED

RTL observers are cycle-accurate and need system clocks to operate. Observers in other abstraction levels may have different requirements.

5.2 Results

For the verification of the traffic light controller, four observers (implementing temporal patterns) from a predefined library were instantiated. Simulation trace in Figure 18 shows the design signals in the upper half and state/event outputs from the adapter in the lower half. Though the first three constraints satisfy this particular execution scenario, the last exclusion relation between the ns_light. GREEN and ew_light.YELLOW fails, as shown by the red marker in the execution trace (circled in the figure). This is exactly the case with this faulty FSM as presented in the book (chapter 7 case study) [44]. To summarize, some of the observations made from this work are:

- This framework makes explicit all the steps between the natural language specification, expressed as a UML diagram, and resulting design verification.
- Notion of adapters is introduced to remove ambiguities between the concepts of states and events.
- States are encoded to represent temporal patterns in behavior. These state-based relations are then transformed into CCSL events. These events are finally encoded as a property used for verification.

6 Expressing Temporal Patterns Formally

This section highlights the second major contribution of this proposed framework. As discussed earlier, all the state-state and state-event relations presented previously can be encoded in CCSL, which in turn can then be used to generate HDL code for the intended relation. But a more direct approach to expressing these operators is to formally encode them as automata directly. Then these automata can easily be encoded in HDLs like Verilog. This proposed framework advocates the direct approach as a fast, less complex alternative to two-step approach through CCSL and TimeSquare tool. Next we discuss the automaton of some of the state/event relations and the derived Verilog code. A comprehensive list of these operator automata is given on the project website [50].

To discuss the automata for the state relations, we start with the less complex **state-event relations**. The *excludes* relation A # e can be viewed as a collection of three distinct interacting events a_s, a_f and e. Figure 19 describes the automaton of exclusion relation of a state and an event. Here the trigger $\langle \epsilon, \epsilon \rangle$ represents a no operation transition which occurs at each state when nothing else is happening. As the state events a_s and a_f are mutually exclusive, the first transition value in $\langle \epsilon, \epsilon \rangle$ represents the state A events, while the second one represents event e. Here the transition $\langle a_f, e \rangle$ is the coincident occurrence of events a_f and e. The automaton given in the figure consists of two states A′ and A representing the current status of state A and a violation state V. Any transition causing violation state is irreversible and in any such scenario system is considered violated and invalid. Similar exclusion behavior can be represented in CCSL using the two relations, as

$$e \searrow as \boxed{\preccurlyeq} af$$
$$as \boxed{\sim} af$$

The first rule samples e on the event a_s and the result is tested to precede the a_f. If e comes in between the a_s and a_f, then the sampling will give wrong results. Here it is important to note that the second CCSL relation is what we call as the 'integrity rule of the state' while the first one establishes the desired exclusion relation. These two CCSL relations are equivalent to our proposed automaton and apparently our direct approach is much effective with regards to code size and ease of use.

The trigger operator in its basic form consists of two state events a_s, a_f and an external trigger e. The desired behavior automaton is simple, easy to follow and quite similar to the excludes relation, as shown

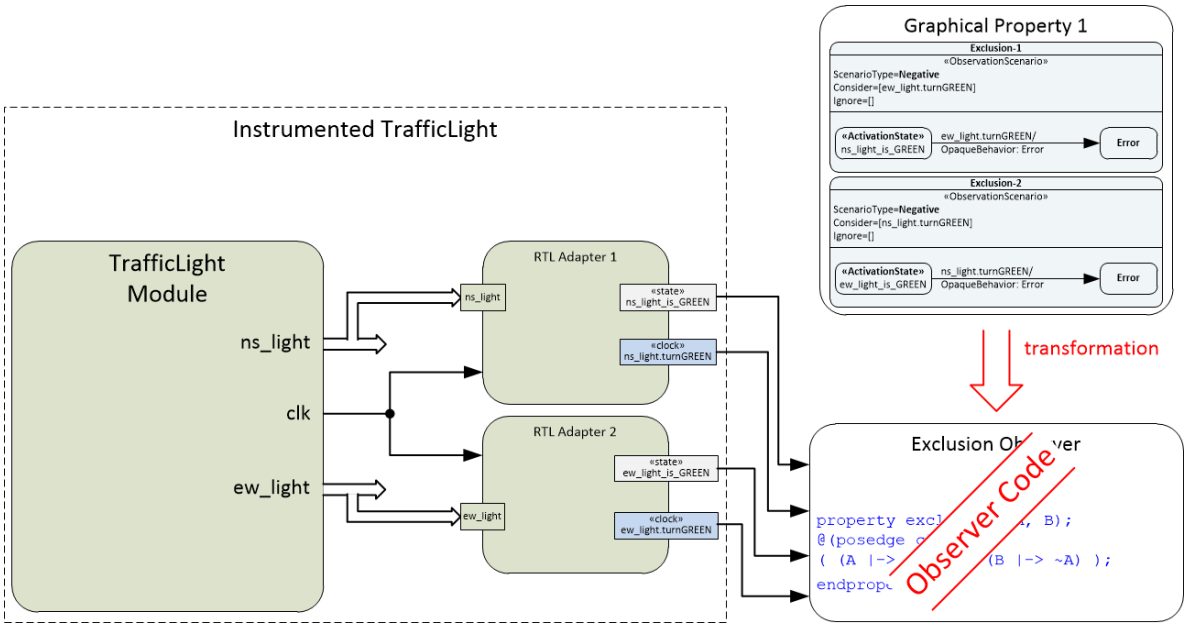


Fig. 17: Integration of Observer in the Verification Environment

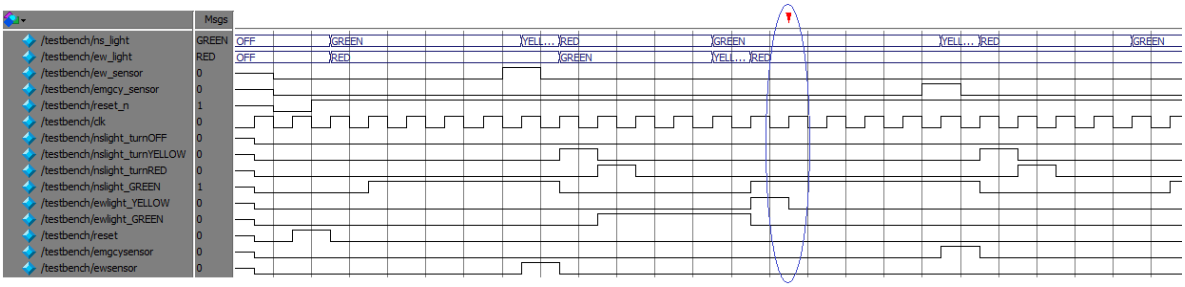


Fig. 18: Simulation Results of the Observers for the Traffic Light Controller

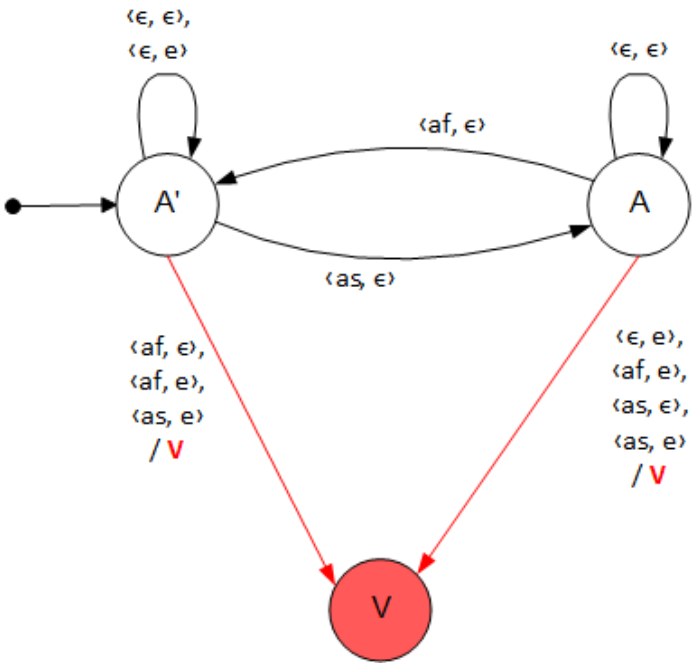


Fig. 19: Automaton of A excludes e

in Figure 20. But when the flexible delayed triggering option is added, the desired automaton explodes quickly (though the number of states are still two) and no more easy to trace just on paper (shown in Figure 21). Several additional variables (ds, df, i, array d[ ] etc) are added for event count and time keeping. Remember to improve the state machine readability the violation transitions are not shown. All the transitions not shown explicitly are violations. Using CCSL the desired pattern can be achieved using,

$$e(min) \rightsquigarrow clk \boxed{\preccurlyeq} as \boxed{\preccurlyeq} e(max) \rightsquigarrow clk$$
$$as \boxed{\sim} af$$

Here again the second equation is the integrity rule for the states while the first one defines the triggering. In this equation, the event e delayed for 'min' clock events on clk is expected to precede the event a_s. Also the event a_s is expected to precede the event e delayed for 'max' number of clock cycles on clk. So we see that a single operator (trigger) of our framework is represented by five operators in CCSL to get the desired behavior.

For the **state-state relations**, the automaton of the most complex state relation *A precedes B by [m,n] on clk* is given in Figure 22. The introduction of the duration constrained between m and n ticks of clock clk raises many corner cases and hence FSM becomes complex. There are four states in the

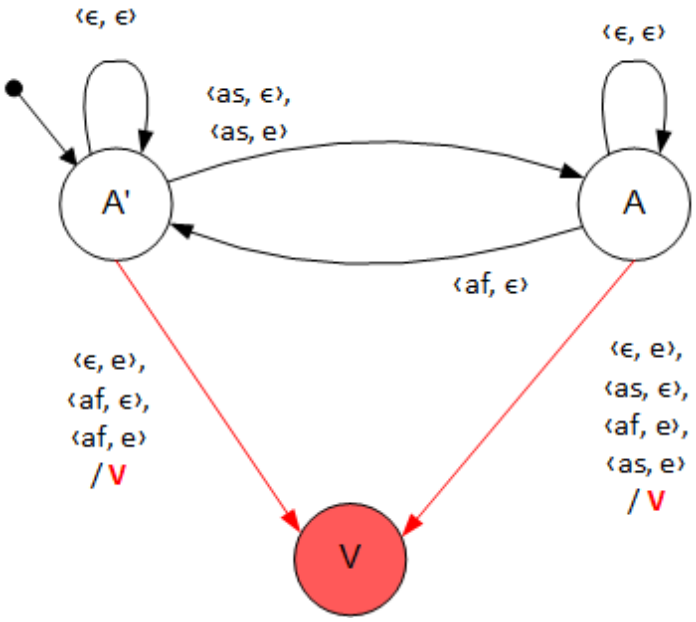


Fig. 20: Automaton of e triggers A

FSM corresponding to the two state variables A and B. A much detailed version of these automata and resulting CCSL and Verilog code is given on the project website [50]. When modeling the same behavior in CCSL we would require several relations,

$$af(min) \rightsquigarrow clk \boxed{\preccurlyeq} bs \boxed{\preccurlyeq} af(max) \rightsquigarrow clk$$
$$as \boxed{\sim} af$$
$$bs \boxed{\sim} bf$$

Here again second and third CCSL relations are the state integrity rules for A and B while the first one is the precedence relation with the delay operator to enforce the limits min and max. If we observe, this relation is quite similar to the one for trigger, as both the operators with limit constraints behave quite similar. The equation tells us that a_f delayed for some min clock cycles should precede b_s and then in the second part b_s must precede the same a_f signal delayed this time for max cycles of clock.

7 Implementation

Figure 23 shows the implementation approach used for the proposed framework. The system modeling can be done in any UML tool but we suggest to use Eclipse-based tools (such as Papyrus UML) as it well incorporates our model transformation. Modeling temporal patterns using state machine diagrams requires Observation profile. For the interaction diagrams, MARTE profile is optionally needed for duration specification.

We have developed a model transformation plugin named TemPAC (Temporal Pattern Analyzer and Code Generator). It is implemented in java based on the Eclipse Modeling Framework (EMF) [54,55]. The model transformation plugin is available online from the project website [56]. The plugin generates three types of code output files: temporal property textual code, its observer as a Verilog code implementation, and the equivalent CCSL properties (as shown in Figure 23). The temporal pattern textual code is based on the state relation operators presented in the earlier sections. These operators are the direct textual description of the graphical temporal properties. The generated Verilog code is the observer for these temporal operators derived directly from the state operator automata described earlier.

Lastly, the transformation plugin also generates the equivalent code in CCSL in the form of CCSL temporal constraints.

Representation of state-based behavior as CCSL constraints provides numerous advantages. The generated CCSL code can be fed to the TimeSquare tool [18] for early validation of the system behavior and simulation of possible scenarios. TimeSquare also provides the facility to generate Verilog observer code. This two-step code generation is equivalent to our EMF-based code generation directly from the model. As an example, we model the exclusion temporal pattern (shown in Figure 8) between the state A and the event e. The state temporal property generated from its model is straightforward textual representation.

```
SM1 = A excludes e
```

The generated Verilog code is the observer for the exclusion operator implemented from the automaton directly.

```
module Exclusion (
  input as,
  input af,
  input e,
  output violation );

  reg [1:0] FSM = 0;
  reg v = 0;

  always @ (as or af or e)
  begin
    case (FSM)
      0 :
        if (as==1'b0 && af==1'b0 && e==1'b0) FSM=0;
        else if (as==1'b0 && af==1'b0 && e==1'b1) FSM=0;
        else if (as==1'b1 && af==1'b0 && e==1'b0) FSM=1;
        else
        begin
          FSM=2;
          v=1'b1;
        end
      1 :
        if (as==1'b0 && af==1'b0 && e==1'b0) FSM=1;
        else if (as==1'b0 && af==1'b1 && e==1'b0) FSM=0;
        else
        begin
          FSM=2;
          v=1'b1;
        end
      default :
        FSM=2;
        v=1'b1;
    endcase
  end

  assign violation = v;
endmodule
```

Lastly the CCSL code generated from the UML using EMF plugin is given next.

```
SM1.as alternatesWith SM1.af
e sampledOn SM1.as precedes SM1.af
```

where the logical clocks SM1.as and SM1.af represents the start and end of the state A respectively. This generated code contains two parts, the relation ensuring state integrity and the relation enforcing the

temporal property. The state integrity rule ensures the state start event alternates with the occurrence of state terminate event.

8 Conclusion and Future Work

This paper presents a natural way to interpret UML diagrams annotated with features from MARTE to specify system requirements. The framework proposed a UML based approach to capture properties in a bid to replace temporal logic properties like in LTL. It also proposed a way to extend the existing capabilities of CCSL which can though represent state relations but is not practically meant for that task. The framework identified two major categories of temporal patterns, state-based and mixed state/event relations. Semantics of the states in both types of properties have been expressed as state-start and state-end events and can be expressed in the form of CCSL specification. This CCSL specification can then be analyzed to detect bugs in the system specification. An exhaustive set of state relations, based on Allen's work, have been proposed. Later these relations are implemented using a subset of state machine diagrams and sequence diagrams coupled with features from MARTE time model. This framework has presented a tool plugin for model transformation of such graphical patterns directly into CCSL and observers based-on Verilog HDL.

The presented work provided a comprehensive outlook on the implementation and use of temporal patterns. However it would be interesting to see application of this approach from the examples of different domains like safety critical systems or automotive embedded systems. Moreover serious attempts can be made to formally incorporate CCSL in the proposed framework. Presently CCSL is used to represent event-event relations in a system. We propose a new unified language CCSL+, an extension of CCSL, that can handle all sorts of state-based, event-based or mixed relations.

Acknowledgements This project is partially funded by NSTIP (National Science Technology and Innovative Plan), Saudi Arabia under the Track "Software Engineering and Innovated Systems" bearing the project code "13-INF761-10".

References

1. P.R. Panda, in *Proceedings of the 14th International Symposium on Systems Synthesis* (ACM, New York, NY, USA, 2001), ISSS '01, pp. 75–80. DOI 10.1145/500001.500018. URL http://doi.acm.org/10.1145/500001.500018
2. S. Rigo, R. Azevedo, L. Santos, *Electronic System Level Design: An Open-Source Approach*, 1st edn. (Springer Publishing Company, Incorporated, 2011)
3. R. Drechsler, M. Soeken, R. Wille, in *Specification and Design Languages (FDL), 2012 Forum on* (2012), pp. 53–58
4. M. Soeken, R. Drechsler, *Formal Specification Level - Concepts, Methods, and Algorithms* (Springer, 2015). DOI 10.1007/978-3-319-08699-6. URL http://dx.doi.org/10.1007/978-3-319-08699-6
5. I. Harris, in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE* (2012), pp. 1252–1253. DOI 10.1145/2228360.2228591
6. J. Rumbaugh, I. Jacobson, G. Booch (eds.), *The Unified Modeling Language Reference Manual* (Addison-Wesley Longman Ltd., Essex, UK, UK, 1999)
7. B. Selic, S. Grard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*, 1st edn. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013)
8. Object Management Group. Time Modeling in UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems (2011). URL http://www.omg.org/spec/MARTE/
9. T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008)
10. C. André, Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA (2009). URL https://hal.inria.fr/inria-00384077
11. C. André, J. Deantoni, F. Mallet, R. De Simone, in *Synthesis of Embedded Software: Frameworks and Methodologies for Correctness by Construction*, ed. by S.K. Shukla, J.P. Talpin (Springer Science+Business Media, LLC 2010, 2010), p. 28. URL https://hal.inria.fr/inria-00495664. Chapter 7
12. A. Pnueli, in *18th Annual Symposium on Foundations of Computer Science* (IEEE Computer Society, 1977), pp. 46–57. DOI 10.1109/SFCS.1977.32. URL http://dx.doi.org/10.1109/SFCS.1977.32
13. E. Clarke, O. Grumberg, D. Peled, *Model Checking* (MIT Press, 1999). URL https://books.google.com.om/books?id=Nmc4wEaLXFEC
14. B. Wile, J. Goss, W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005)
15. M. Chai, B.H. Schlingloff, *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings* (Springer International Publishing, Cham, 2014), chap. Monitoring Systems with Extended Live Sequence Charts, pp. 48–63. DOI 10.1007/978-3-319-11164-3_5. URL http://dx.doi.org/10.1007/978-3-319-11164-3_5

16. IEEE. Property Specification Language (PSL) (2010). URL http://standards.ieee.org/findstds/standard/1850-2010.html
17. P.L. Guernic, T. Gautier, J. Talpin, L. Besnard, in *2015 International Symposium on Theoretical Aspects of Software Engineering, TASE 2015* (IEEE Computer Society, 2015), pp. 95–102. DOI 10.1109/TASE.2015.21. URL http://dx.doi.org/10.1109/TASE.2015.21
18. J. Deantoni, F. Mallet, in *TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012*, vol. 7304, ed. by S.N. Carlo A. Furia. Czech Technical University in Prague, in co-operation with ETH Zurich (Springer, Prague, Czech Republic, 2012), vol. 7304, pp. 34–41. DOI 10.1007/978-3-642-30561-0_4
19. F. Mallet, R. de Simone, Sci. Comput. Program. **106**, 78 (2015). DOI 10.1016/j.scico.2015.03.001. URL http://dx.doi.org/10.1016/j.scico.2015.03.001
20. A.M. Khan, F. Mallet, M. Rashid, Natural Interpretation of UML/MARTE Diagrams for System Requirements Specification [Under Review] (2016)
21. M.B. Dwyer, G.S. Avrunin, J.C. Corbett, in *Proceedings of the 21st International Conference on Software Engineering* (ACM, New York, NY, USA, 1999), ICSE '99, pp. 411–420. DOI 10.1145/302405.302672. URL http://doi.acm.org/10.1145/302405.302672
22. R.L. Smith, G.S. Avrunin, L.A. Clarke, L.J. Osterweil, in *Proceedings of the 24th International Conference on Software Engineering* (ACM, New York, NY, USA, 2002), ICSE '02, pp. 11–21. DOI 10.1145/581339.581345. URL http://doi.acm.org/10.1145/581339.581345
23. L. Zanolin, C. Ghezzi, L. Baresi. An approach to model and validate publish/subscribe architectures (2003)
24. A. Alfonso, V. Braberman, N. Kicillof, A. Olivero, in *Proceedings of the 26th International Conference on Software Engineering* (IEEE Computer Society, Washington, DC, USA, 2004), ICSE '04, pp. 168–177. URL http://dl.acm.org/citation.cfm?id=998675.999423
25. H. Kugler, D. Harel, A. Pnueli, Y. Lu, Y. Bontemps, in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Springer-Verlag, Berlin, Heidelberg, 2005), TACAS'05, pp. 445–460. DOI 10.1007/978-3-540-31980-1_29. URL http://dx.doi.org/10.1007/978-3-540-31980-1_29
26. P. Zhang, L. Grunske, A. Tang, B. Li, in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (IEEE Computer Society, Washington, DC, USA, 2009), ASE '09, pp. 500–504. DOI 10.1109/ASE.2009.56. URL http://dx.doi.org/10.1109/ASE.2009.56
27. P. Zhang, B. Li, L. Grunske, J. Syst. Softw. **83**(3), 371 (2010). DOI 10.1016/j.jss.2009.09.013. URL http://dx.doi.org/10.1016/j.jss.2009.09.013
28. M. Autili, P. Inverardi, P. Pelliccione, Automated Software Engg. **14**(3), 293 (2007). DOI 10.1007/s10515-007-0012-6. URL http://dx.doi.org/10.1007/s10515-007-0012-6
29. M. Autili, P. Pelliccione, Electron. Notes Theor. Comput. Sci. **211**, 147 (2008). DOI 10.1016/j.entcs.2008.04.037. URL http://dx.doi.org/10.1016/j.entcs.2008.04.037
30. R. Gascon, F. Mallet, J. Deantoni, in *Proceedings of the 2011 Eighteenth International Symposium on Temporal Representation and Reasoning* (IEEE Computer Society, Washington, DC, USA, 2011), TIME '11, pp. 141–148. DOI 10.1109/TIME.2011.10. URL http://dx.doi.org/10.1109/TIME.2011.10
31. S. Konrad, B.H.C. Cheng, in *Proceedings of the 27th International Conference on Software Engineering* (ACM, New York, NY, USA, 2005), ICSE '05, pp. 372–381. DOI 10.1145/1062455.1062526. URL http://doi.acm.org/10.1145/1062455.1062526
32. L. Di Guglielmo, F. Fummi, N. Orlandi, G. Pravadelli, in *Computer Design (ICCD), 2010 IEEE International Conference on* (2010), pp. 468–473. DOI 10.1109/ICCD.2010.5647654
33. M. Leucker, C. Schallhart, The Journal of Logic and Algebraic Programming **78**(5), 293 (2009). DOI http://dx.doi.org/10.1016/j.jlap.2008.08.004. URL http://www.sciencedirect.com/science/article/pii/S1567832608000775. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07)
34. R.V. Community. Runtime verification events 2001-16 (2016). URL http://runtime-verification.org/
35. A. Bauer, M. Leucker, C. Schallhart, ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1 (2011). DOI 10.1145/2000799.2000800. URL http://doi.acm.org/10.1145/2000799.2000800
36. C. Watterson, D. Heffernan, IET Software **1**(5), 172 (2007). DOI 10.1049/iet-sen:20060076
37. H. Yu, J.P. Talpin, L. Besnard, T. Gautier, H. Marchand, P.L. Guernic, in *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on* (2011), pp. 21–30. DOI 10.1109/MEMCOD.2011.5970507
38. M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, A. Tang, IEEE Transactions on Software Engineering (TSE) **41**(7), 620 (2015). DOI 10.1109/TSE.2015.2398877
39. Object Management Group (OMG). Unified Modeling Language (UML), Superstructure Specification, Version 2.4 (2011)
40. C. Andr, F. Mallet, R. de Simone, in *Model Driven Engineering Languages and Systems*, *Lecture Notes in Computer Science*, vol. 4735, ed. by G. Engels, B. Opdyke, D. Schmidt, F. Weil (Springer Berlin Heidelberg, 2007), pp. 559–573. DOI 10.1007/978-3-540-75209-7_38. URL http://dx.doi.org/10.1007/978-3-540-75209-7_38
41. J.R. Abrial, E. Börger, H. Langmaack, in *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the Book Grow out of a Dagstuhl Seminar, June 1995).* (Springer-Verlag, London, UK, UK, 1996), pp. 1–12. URL http://dl.acm.org/citation.cfm?id=647370.723887
42. M. Al-Lail, W. Sun, R.B. France, in *Quality Software (QSIC), 2014 14th International Conference on* (2014), pp. 196–201. DOI 10.1109/QSIC.2014.56
43. A.E. Haxthausen, Int. J. Softw. Tools Technol. Transf. **16**(6), 713 (2014). DOI 10.1007/s10009-013-0295-9. URL http://dx.doi.org/10.1007/s10009-013-0295-9
44. B. Cohen, S. Venkataramanan, A. Kumari, L. Piper, *SystemVerilog Assertions Handbook: for Dynamic and Formal Verification*, 2nd edn. (VhdlCohen Publishing, Palos Verdes Peninsula, CA, USA, 2010)

45. J.F. Allen, Commun. ACM **26**(11), 832 (1983). DOI 10.1145/182.358434. URL http://doi.acm.org/10.1145/182.358434
46. J. Suryadevara, Model based development of embedded systems using logical clock constraints and timed automata. Ph.D. thesis, Malardalen University, Sweden (2013)
47. F. Mallet, *Formal Modeling and Verification of Cyber-Physical Systems: 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015* (Springer Fachmedien Wiesbaden, Wiesbaden, 2015), chap. MARTE/CCSL for Modeling Cyber-Physical Systems, pp. 26–49. DOI 10.1007/978-3-658-09994-7_2. URL http://dx.doi.org/10.1007/978-3-658-09994-7_2
48. F. Mallet, Innovations in Systems and Software Engineering **4**(3), 309 (2008). DOI 10.1007/s11334-008-0055-2. URL http://dx.doi.org/10.1007/s11334-008-0055-2
49. F. Mallet, C. André, Uml/marte ccsl, signal and petri nets. Research Report RR-6545, INRIA (2008). URL https://hal.inria.fr/inria-00283077v4
50. A.M. Khan, Semantics of Graphical Temporal Patterns using UML/MARTE. Research report, NSTIP. URL http://www.modeves.com/patterns.html
51. N. Halbwachs, F. Lagnier, P. Raymond, in *Algebraic Methodology and Software Technology (AMAST93)*, ed. by M. Nivat, C. Rattray, T. Rus, G. Scollo, Workshops in Computing (Springer London, 1994), pp. 83–96. DOI 10.1007/978-1-4471-3227-1_8. URL http://dx.doi.org/10.1007/978-1-4471-3227-1_8
52. L. Aceto, A. Burgueo, K. Larsen, in *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science*, vol. 1384, ed. by B. Steffen (Springer Berlin Heidelberg, 1998), pp. 263–280. DOI 10.1007/BFb0054177. URL http://dx.doi.org/10.1007/BFb0054177
53. S. Bensalem, M. Bozga, M. Krichen, S. Tripakis, Electronic Notes in Theoretical Computer Science **113**, 23 (2005). DOI http://dx.doi.org/10.1016/j.entcs.2004.01.036. URL http://www.sciencedirect.com/science/article/pii/S157106610405251X. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004)Fourth Workshop on Runtime Verification 2004
54. D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd edn. (Addison-Wesley Professional, 2009)
55. The Eclipse Foundation. Eclipse Modeling Framework (EMF). URL http://www.eclipse.org/modeling/emf/
56. A.M. Khan. TemPAC: Temporal Pattern Analyzer and Code-generator EMF plugin. URL http://www.modeves.com/tempac.html

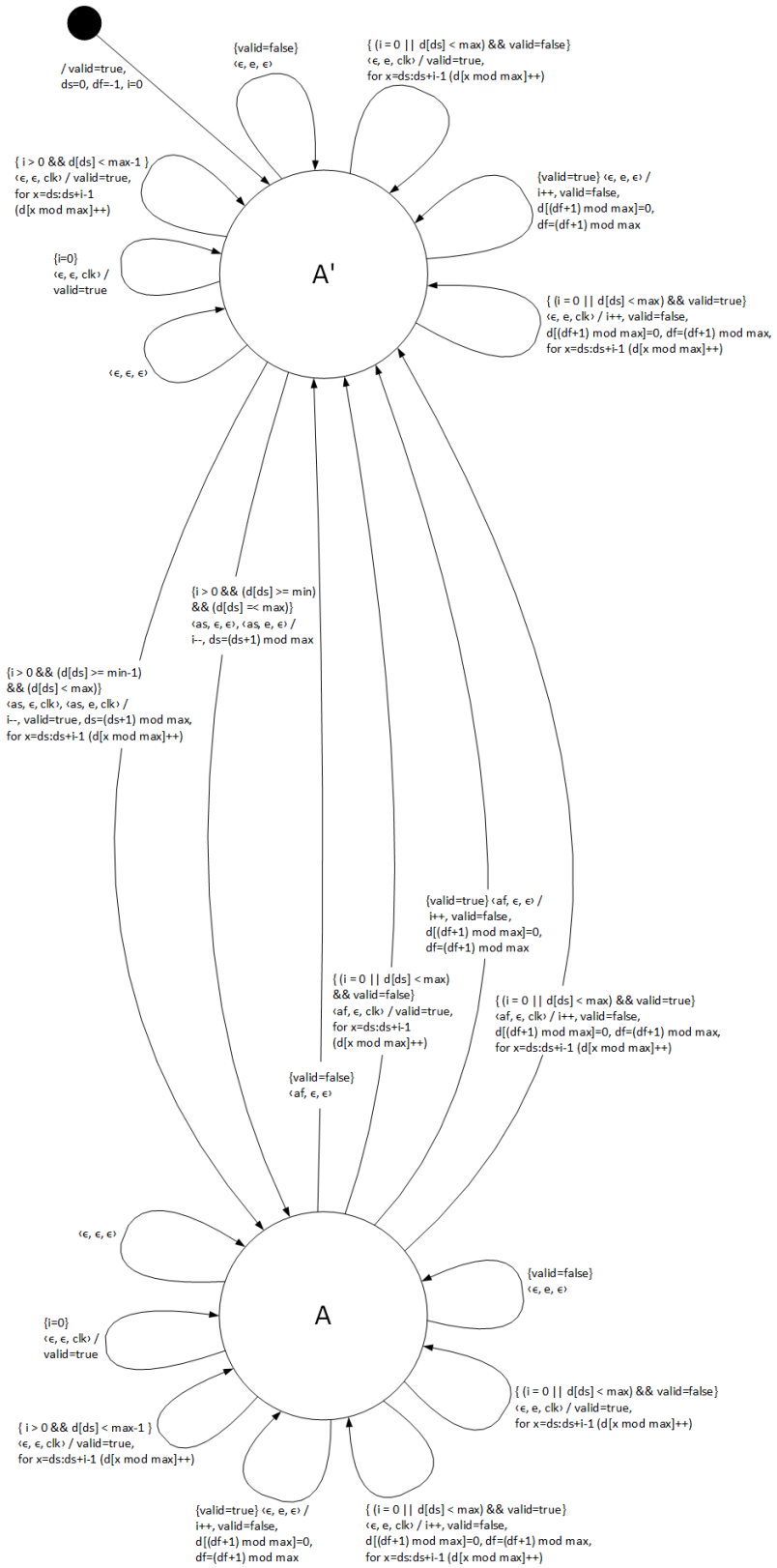


Fig. 21: Automaton of e triggers A after [m,n] on clk

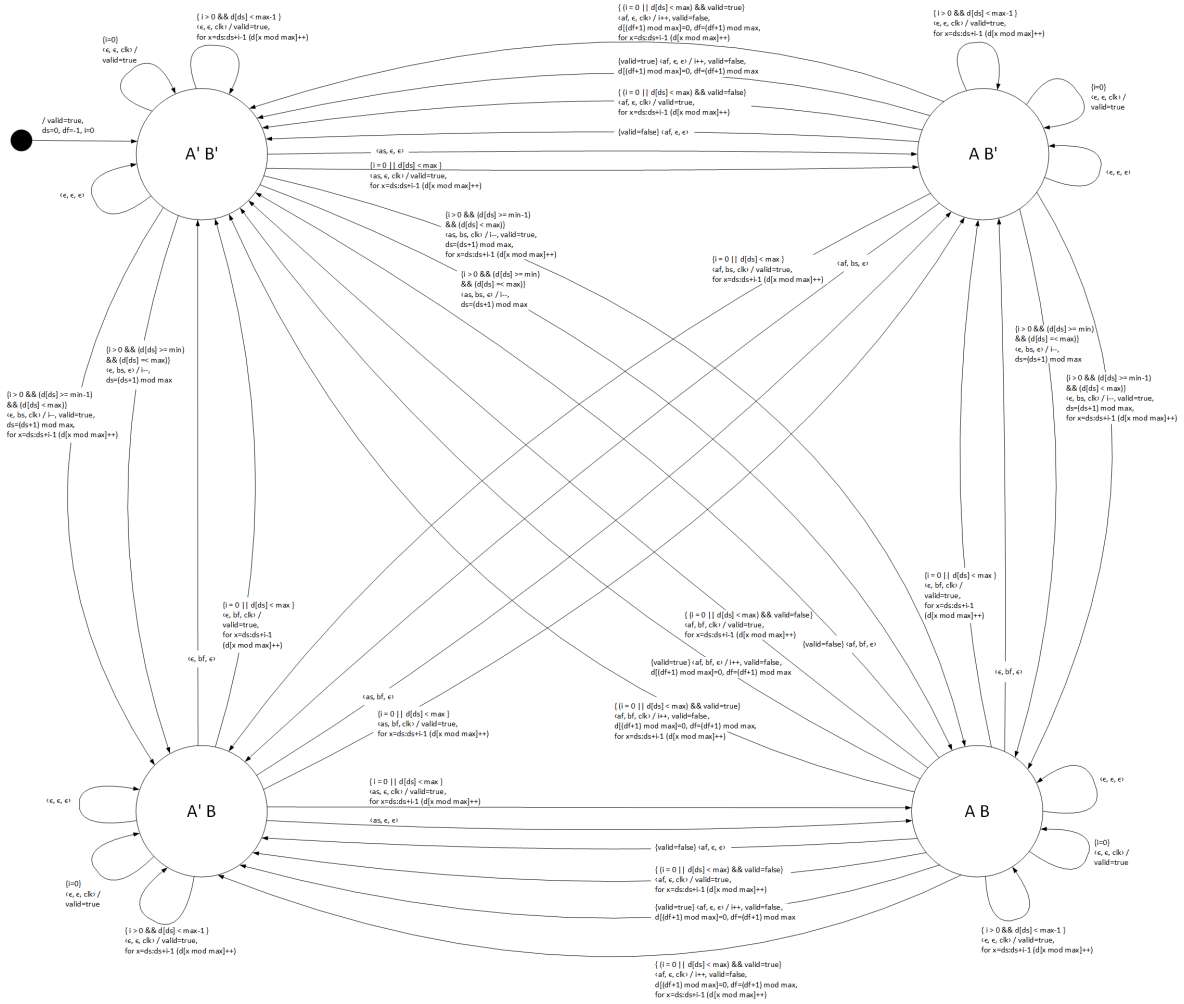


Fig. 22: Automaton of A precedes B by [m,n] on clk

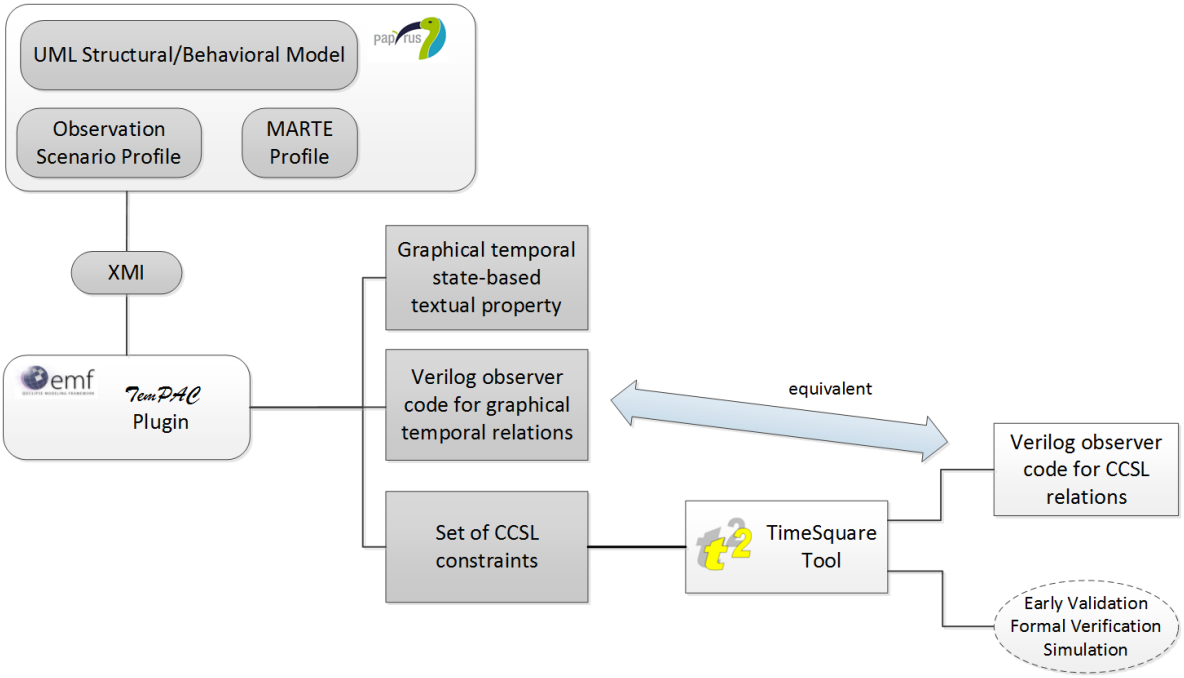


Fig. 23: Framework Work-flow and Model transformation